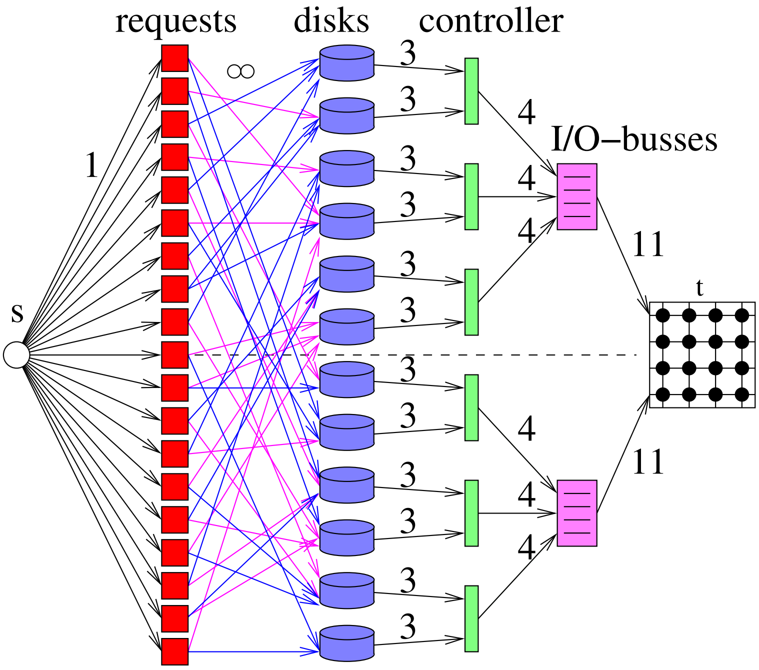
Algorithmen II (WS 17/18)
Course Notes

Timo Bingmann, Johannes Fischer, Robert Geisberger, Moritz Kobitzsch, Vitaly Osipov, Peter Sanders, Dennis Schieferdecker, Christian Schulz, Johannes Singler

Institute of Theoretical Informatics, Algorithmics II
Karlsruhe Institute of Technology (KIT)



Last modified: December 4, 2017

Contents

Preface

The present course notes collect and elaborate on material for the bachelor course “Algorithmen II” at KIT. They were first published in winter term 2010/2011 and have been expanded since then. Nevertheless, the current state of the course notes is still preliminary – in software jargon we would call it an α-version. Also, take note that the topics presented in here sometimes go beyond what was presented during the lectures. This aditional material is not required to successfully complete the course, it is given to broaden your algorithmic horizon.

Although the lecture is presented in German, the notes are written in English as they are largely compiled from papers or textbooks written in English. Hence, note that little here is original material and kindly follow the citations to the original sources for more detailed information.

The chapters in these notes directly correspond to the chapters on the lecture slides. For chapters 2–4 on *Advanced Data Structures*, *Shortest Paths* and *Applications of DFS* we refer to the textbook "Algorithms and Data Structures: The Basic Toolbox" by Kurt Mehlhorn and Peter Sanders [62], that is also available online and for free.

Chapter 1

Algorithm Engineering

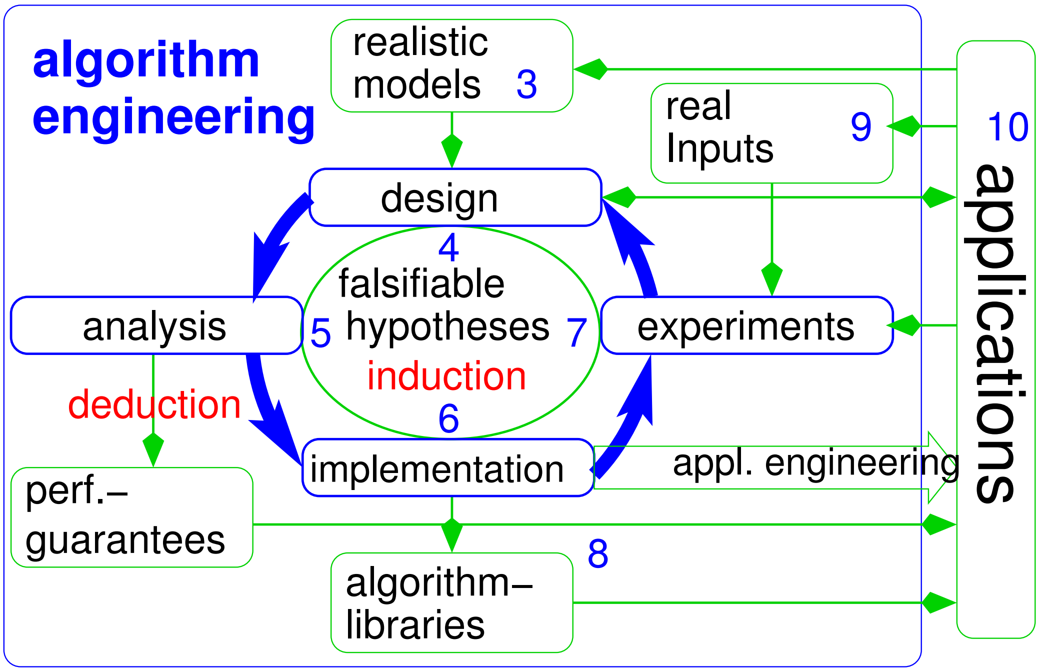


Figure 1.1: Algorithm engineering as a cycle of design, analysis, implementation, and experimental evaluation driven by falsifiable hypotheses. The numbers refer to sections.

This is a shortened version of [78]. 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 [86]. Rather than just processing the data coming our 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,...

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 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 [71]. 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. 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 poissible 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 1.1 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.

1.1 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 [30], and several overview papers on the subject were published [2, 65, 54, 55, 44].

The term "algorithm engineering" already appears 1986 in the Foreword of [9] and 1989 in the title of [84]. 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.

[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 conjuction with SODA. WEA, now SEA was first organized in 2002.

P. Italiano organized the “Workshop on Algorithm Engineering” in 1997 and also uses “algorithm engineering” as the title for the algorithms column of EATCS in 2003 [24] 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 [2]. 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.

1.2 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) [1, 87, 63] which is a careful refinement of the von Neumann model [67]. Instead of a uniform memory, there are two levels of memory. A fast memory of limited size M and 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 [59], 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 example for application modelling is the simulation of traffic flows. While microscopic simulations that take the actual behavior of every car into account

[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.

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.

1.3 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 [34] is much worse than theoretically inferior algorithms [17, 58].

1.4 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. [89]) 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., [32] 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 inpracticable 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 [83]. 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 [7]. 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.

1.5 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 $\mathrm{O}(nm\log n)$ matching algorithm in [60]. Its abstract description requires a sophisticated data structure whose efficient implementation only succeeded in [60].

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 [15, 8, 6].

Even the implementation of relatively simple basic algorithms can be challenging. You often have to compare several candidates based on small constant 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. [77, 81],[13]).

Often only implementations give convincing evidence of the correctness and result quality of an algorithm. For example, an algorithm for planar embedding [39] 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. [56]). This was only noticed during the first correct implementation [57]. Similarly, for a long time nobody suceeded in implementing the famous algorithm for computing three-connected components from [42]. Only an implementation in 2000 [37] 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 [45].

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.

1.6 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 [4] 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 [79].

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.

1.7 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 reimplementation, 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 /$\cdots$ 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 [58] 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 [22]) or multicore parallelism (MCSTL, GNU C++ standard library [82]). 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.

1.8 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 [3], 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 [5] and of the Steiner tree problem [70]. 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 [80] enabling a revolution with speedups of up to six orders of magnitude over Dijkstra's algorithm and a perspective for many applications [21]. 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

[3] `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`

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.

1.9 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.

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,...

1.10 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.

Chapter 5

Maximum Flows and Matchings

The *maximum flow problem* is very easy to state: In a capacitated network, we wish to send as much flow as possible between two special nodes, a source node s and a sink node t, without exceeding the capacity of any edge. In this chapter we discuss a number of algorithms for solving the maximum flow problem. These algorithms are of two types:

1. **Augmenting path algorithms** which incrementally augment flow along paths from the source node to the sink node.

2. **Preflow-push algorithms** that flood the network so that some nodes have excesses (or buildup of flow). These algorithms incrementally relieve flow from nodes with excesses by sending flow from the node forward toward the sink node or backward toward the source node.

Definition 1
We define a network *to be a directed weighted graph $G = (V, E)$ with distinct vertices s and t in G and the restriction that s has no incoming edges and t has no outgoing edges. We call s a* source node *and t a* sink node. *For an edge e, we call its weight c_e the* capacity *of e (nonnegative!). An (s,t)*-flow *or simply* flow *is a function $f : E \mapsto \mathbb{R}_{\geq 0}$ ($f_e := f(e)$) satisfying the capacity constraints and the flow conservation constraints:*

1. $0 \leq f_e \leq c_e \; \forall e \in E$

2. $\sum_{e:\text{source}(e)=v} f_e = \sum_{e:\text{target}(e)=v} f_e \; \forall v \in V \backslash \{s, t\}$

The capacity constraints state that the flow across any edge is bounded by its capacity, and the latter constraints state that for each node different from s and t, the total incoming flow equals the total outgoing flow. We further define the value of a flow **val**(f) *= total outgoing flow from s. Our goal is to find a flow with maximum value. An example can be found in Figure 5.1.*

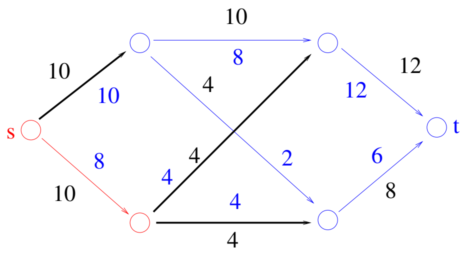


Figure 5.1: An example network in which capacities are marked black and the current flow values on an edge are marked blue. The total flow value is 18 which is maximal. We also see a cut in the network induced by the colors.

5.1 Algorithms 1956–now

Algorithms for the Maximum Flow Problem have been studied for over five decades now. Starting with early work in linear programming and spurred by the classic book of Ford and Fulkerson, the study of this problem has led to continuing improvements in the efficiency of network flow algorithms. We give a brief overview in the following table. Here n is the number of nodes, m the number of arcs/edges and U the largest capacity in the graph.

Year	Author	Running time
1956	Ford-Fulkerson	$O(mnU)$
1969	Edmonds-Karp	$O(m^2n)$
1970	Dinic	$O(mn^2)$
1973	Dinic-Gabow	$O(mn\log U)$
1974	Karzanov	$O(n^3)$
1977	Cherkassky	$O(n^2\sqrt{m})$
1980	Galil-Naamad	$O(mn\log^2 n)$
1983	Sleator-Tarjan	$O(mn\log n)$
1986	Goldberg-Tarjan	$O(mn\log(n^2/m))$
1987	Ahuja-Orlin	$O(mn + n^2\log U)$
1987	Ahuja-Orlin-Tarjan	$O(mn\log(2+n\sqrt{\log U}/m))$
1990	Cheriyan-Hagerup-Mehlhorn	$O(n^3/\log n)$
1990	Alon	$O(mn + n^{8/3}\log n)$
1992	King-Rao-Tarjan	$O(mn + n^{2+e})$
1993	Philipps-Westbrook	$O(mn\log n/\log\frac{m}{n} + n^2\log^{2+\varepsilon} n)$
1994	King-Rao-Tarjan	$O(mn\log n/\log\frac{m}{n\log n})$ if $m \geq 2n\log n$
1997	Goldberg-Rao	$O(\min\{m^{1/2}, n^{2/3}\}m\log(n^2/m)\log U)$

5.2 Augmenting Path Algorithms

The basic idea behind augmenting path algorithms is to iteratively identify a path from s to t such that each edge has some *spare capacity*. On this path, the edges with the smallest spare capacity are *saturated*, i.e. the flow along this path is incremented by this capacity. This way the flow is incrementally augmented along paths from the source node to the sink node. To be more precise we need the definition of a residual graph:

Definition 2
Given a network $G = (V, E, c)$ and a flow f, we define the residual network to be $G_f = (V, E_f, c^f)$. The residual network G_f with respect to a flow f has the same node set as G. Every edge of G_f is induced by an edge of G and has so-called residual capacity. Let $e = (u, v)$ be an arbitrary edge of G. If $f(e) < c_e$ then e is also an edge of G_f. Its residual capacity is $c_e^f = c_e - f(e)$. If $f(e) > 0$ then $e^{\text{rev}} = (v, u)$ is an edge of G_f. Its residual capacity is $c_{e^{\text{rev}}}^f = f(e)$. Note that both can be the case, i.e. $e \in E_f$ and $e^{\text{rev}} \in E_f$. An example is given in Figure 5.2.
In other words, the residual capacity $c_{e=(u,v)}^f$ of an edge e is the maximum additional flow that can be sent from node u to node v using the edges (u, v) and (v, u). Sending flow from u to v via the edge (v, u) means to decrease the flow that is on this edge, i.e. decreasing the flow from v to u.

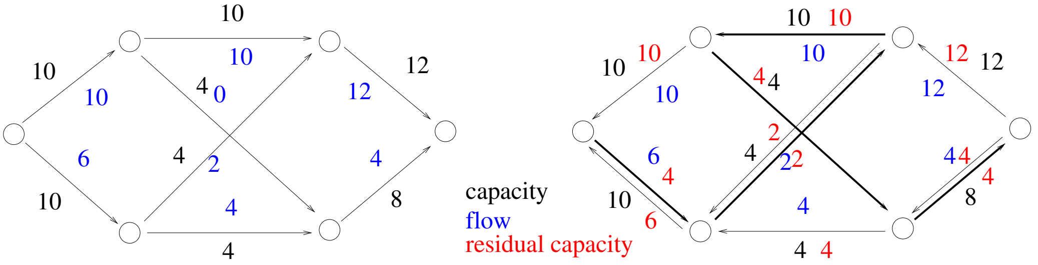


Figure 5.2: On the left hand side we see the original graph with a flow f shown. On the right hand side we see the corresponding residual network. In this graph the highlighted path is an augmenting path with residual capacity $\Delta f = \min\{4, 2, 10, 4, 4\} = 2$.

5.2.1 The Ford Fulkerson Algorithm

We now describe one of the simples and most intuitive algorithms for solving the maximum flow problem. This algorithm is known as the *augmenting path algorithm/Ford Fulkerson algorithm*. We refer to a directed path from the source to the sink in the residual network as an *augmenting path*. We define the residual capacity Δf of an augmenting path p as the minimum residual capacity of any edge in the path. An example for an augmenting path can be found in Figure 5.2. Observe that, by definition, the capacity of an augmenting path is always positive. Consequently, whenever the network contains an augmenting path, we can send additional flow from the source to the sink. The augmenting path algorithm is essentially based on this simple observation. The algorithm proceeds by identifying augmenting paths and augmenting flows on these path until the network contains no such path. Pseudocode for this algorithm can be found in Figure 5.3 and 5.4. For integer capacities, the algorithm runs in $\mathrm{O}(m\mathsf{val}(f))$ time. An example run of the algorithm can be found in Figure 5.5.

```
Function FFMaxFlow($G = (V, E), s, t, \mathsf{c} : E \to \mathbb{N}$) : $E \to \mathbb{N}$
    $f := 0$
    while $\exists$path $p = (s, \dots, t)$ in $G_f$ do
        augmentPath$(G, s, t, c, f)$ // augment $f$ along $p$
    return $f$
```

Figure 5.3: The Ford Fulkerson Augmenting Path algorithm.

```
Procedure augmentPath($G = (V, E), s, t, \mathsf{c} : E \to \mathbb{N}, \mathsf{f} : E \to \mathbb{N}$, path $p$ in $G_f$)
    $\Delta f := \min_{e \in p} c_e^f$
    foreach $(u, v) \in p$ do
        if $(u, v) \in E$ then $f_{(u,v)} \mathrel{+}= \Delta f$
        else $f_{(v,u)} \mathrel{-}= \Delta f$
```

Figure 5.4: The Ford Fulkerson subroutine for augmenting a path.

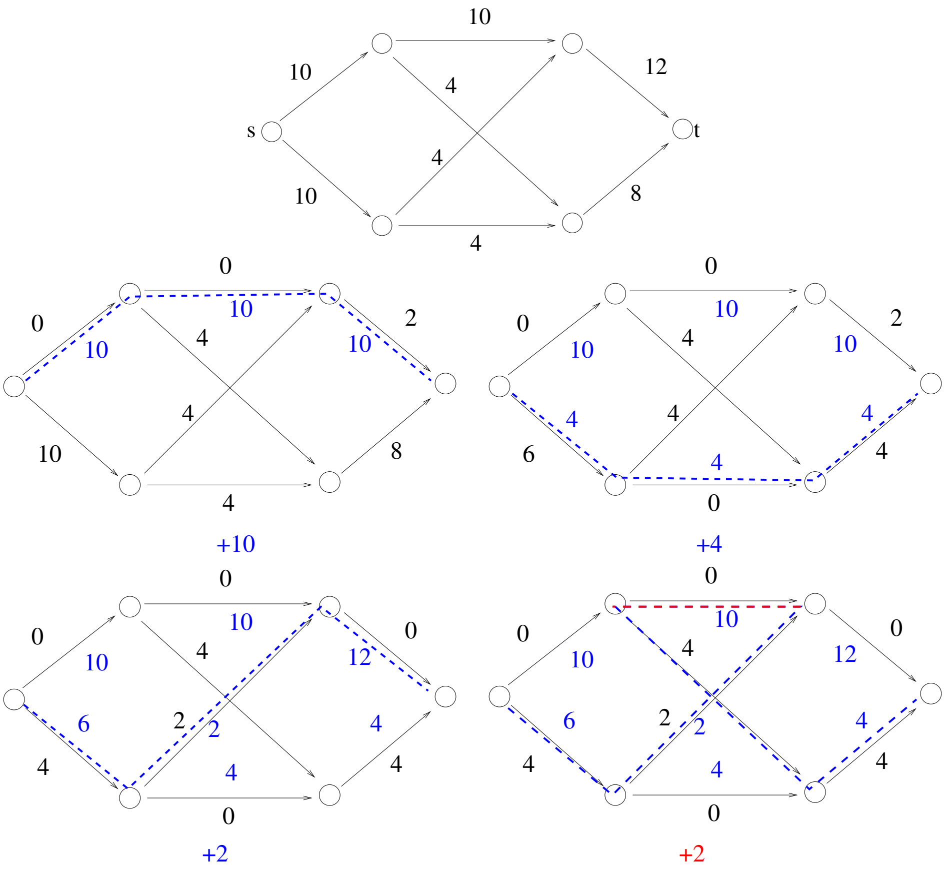


Figure 5.5: An example execution of the augmenting path algorithm.

We now proof that the Ford Fulkerson algorithm is correct.

Definition 3
*An s-t-**cut** is a partition of V into S and $T = V \backslash S$ with $s \in S$ and $t \in T$. The capacity of this cut is the sum of the weights of all edges starting in S and ending in T:*

$$c(S) := c(S, T = V \backslash S) := \sum_{u \in S, v \in T} c_{(u,v)}$$

A cut S is called saturated if $f(e) = c_e$ for all $e = (u, v), u \in S, v \in T$ and $f(e) = 0$ for all $e = (v, u), v \in T, u \in S$.

The next lemma relates flows and cuts: the capacity of any (s,t)-cut is an upper bound for the value of any (s,t)-flow. Conversely, the value of any (s,t)-flow is a lower bound for the capacity of any (s,t)-cut.

Lemma 1
Let f be any $(s,t)-$flow and let S be any (s,t)-cut. Then

$$val(f) \leq c(S).$$

If S is saturated then val$(f) = c(S)$.

Proof.
We have

$$\text{val}(f) = \overbrace{\sum_{e \in E \cap S \times T} f_e}^{S \to T \text{ edges}} - \overbrace{\sum_{e \in E \cap T \times S} f_e}^{T \to S \text{ edges}} \leq \sum_{e \in E \cap S \times T} f_e \leq \sum_{e \in E \cap S \times T} c_e = c(S).$$

For a saturated cut, the inequalities are equalities.

A saturated cut proves the maximality of given flow f. However a saturated cut is easily extracted from a maximum flow by means of the residual network as we will see now.

Lemma 2
Let f be a flow, let G_f be the residual network with respect to f, and let $S := \{v \in V : v$ reachable from s in $G_f\}$ be the set of nodes that are reachable from s in G_f.

1. *If $t \in S$ then f is not maximum.*
2. *If $t \notin S$ then S is a saturated cut and f is maximum.*

Proof.

1. Let p be any simple path from s to t in G_f and let Δf be the minimum residual capacity of any edge of p. Then $\Delta f > 0$. We construct a flow f' of value val(f)+Δf.

$$f'(e) = \begin{cases} f(e) + \Delta f & \text{if } e \text{ is in } p \\ f(e) - \Delta f & \text{if } e^{\text{rev}} \text{ is in } p \\ f(e) & \text{if neither } e \text{ nor } e^{\text{rev}} \text{ belongs to p.} \end{cases}$$

   Then f' is a flow and val(f') =val$(f) + \Delta f$. Note that f' is well-defined because the path is simple.

2. There is no edge (v,w) in G_f with $v \in S$ and $w \in T$. Hence, $f(e) = c(e)$ for any $e \in E \cap (S \times T)$ and $f(e) = 0$ for any $e \in E \cap (T \times S)$, i.e. the cut S is saturated. Thus f is maximal.

Lemma 3
The algorithm of Ford and Fulkerson is correct.

Proof.
Clearly the algorithm computes a feasible flow f (invariant). At termination of the algorithm, we have no augmenting path left in G_f. We further have $\forall(u,v) \in E \cap (S \times T) : c^f_{(u,v)} = 0$ and hence $f_{(v,u)} = 0$. That means that S is saturated and hence S proofs the maximality of f.

Theorem (*Elias/Feinstein/Shannon, Ford/Fulkerson 1956*)
The value of an s-t max-flow f equals the minimum capacity of an s-t-cut.

5.2.2 Two Bad Examples for Ford Fulkerson

We mentioned above that the running time for the Ford Fulkerson algorithm is $\mathrm{O}(m\mathrm{val}(f))$ for integers. However the number of augmenting steps is also dependent of the maximum difference capacity $C := \max_{e\in E} c_e - \min_{e\in E} c_e$ (see Figure 5.6). It even gets worse since for irrational numbers it is possible that the algorithm does not terminate at all (see Figure 5.7).

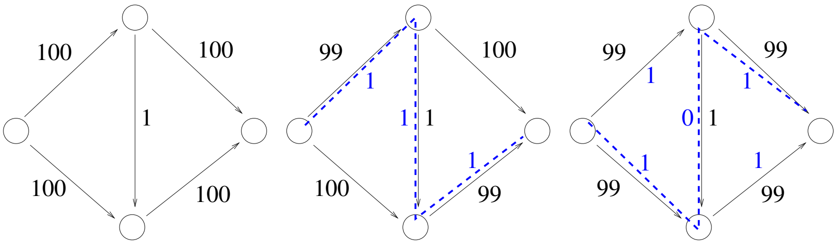


Figure 5.6: By iteratively augmenting the flow along the two blue paths, we get 200 =val(f) path augmentations.

5.2.3 Dinic's Algorithm

There are several maximum flow augmenting path algorithms that send flow along shortest paths from the source to the sink. Dinic's algorithm is a popular algorithm in this class. This algorithm constructs shortest path networks, called *layered networks*, and establishes *blocking flows* (to be defined later) in these networks. In this section we point out the relationship between layered networks and distance labels.

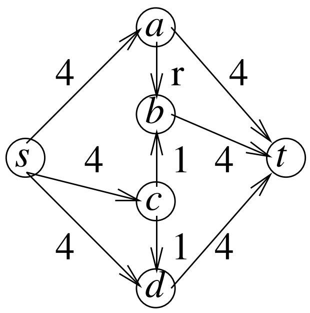


Figure 5.7: Here r is $\frac{\sqrt{5}-1}{2}$. Let $p_0 = \langle s,c,b,t\rangle$, $p_1 = \langle s,a,b,c,d,t\rangle$, $p_2 = \langle s,c,b,a,t\rangle$, $p_3 = \langle s,d,c,b,t\rangle$. Then the sequence of augmenting paths $p_0(p_1,p_2,p_1,p_3)^*$ is an infinite sequence of positive flow augmentations and the flow value does *not* converge to the maximum value 9.

Definition 4 (Distance Labels)
A distance function $d : V \to \mathbb{N} \cup \{0\}$ *with respect to the residual capacities* c_e^f *is a function from the set of nodes to the set of nonnegative integers. We say that a distance function is* valid *with respect to a flow* f *if it satisfies the following two conditions:*

1. $d(t) = 0$

2. $d(u) \leq d(v) + 1$ *for every edge* $(u, v) \in G_f$

We refer to $d(u)$ *as the* distance label *of node* u *and the conditions above as the* validity conditions. *The following properties show why the distance labels might be of use in designing network flow algorithms.*

Property 1
If the distance labels are valid, the distance label $d(u)$ *is a lower bound on the length of the shortest (directed) path from node* u *to node* t *in the residual network.*

Property 2
If $d(s) \geq n$*, the residual network contains no directed path from the source node to the sink node.*

We now introduce some additional notation. We say that the distance labels are *exact* if for each node u, $d(u)$ equals the length of the shortest path from node u to node t in the residual network. We can determine exact distance labels for all nodes in $O(m)$ time by performing a backward breadth-first search (*reverseBFS*) of the network starting at the sink node t.

With respect to a given flow f, we can now define the layered network L_f as follows. First we determine the exact distance labels d in G^f. Then we compute

```
Function DinitzMaxFlow($G = (V, E), s, t, \mathsf{c} : E \to \mathbb{N}$) : $E \to \mathbb{N}$
    $f$:= 0
    while $\exists$path $p = (s, \ldots, t)$ in $G_f$ do
        $d$=$G_f$.reverseBFS($t$) : $V \to \mathbb{N}$
        $L_f = (V, \{(u, v) \in E_f : d(v) = d(u) - 1\})$          // layer graph
        find a blocking flow $f_b$ in $L_f$
        augment $f$ += $f_b$
    return $f$
```

Figure 5.8: Pseudocode for Dinic's algorithm.

the layered network which consists of those edges (u, v) in G^f that satisfy the condition $d(u) = d(v) + 1$. Observe that by definition every path from the source to the sink in the layered network L_f is a shortest path in G^f. Observe further that some edge in L_f might not be contained in any path from the source to the sink. Dinic's algorithm now computes a blocking flow f_b in L_f.

Definition 5
A blocking flow f_b in L_f is a flow such that for each path $p = \langle s, ..., t\rangle$ in L_f there exists an saturated edge $e \in p$, i.e. $f_b(e) = c^f(e)$.

When a blocking flow f_b has been established in a network, Dinic's algorithm adds f_b to f, recomputes the exact distance labels, forms a new layered network, and repeats these computations. The algorithm terminates, when as it is forming the new layered networks, it finds that the source is not connected to the sink. It is possible to show that every time Dinic's algorithm forms a new layered network, the distance label of the source node strictly increases. Consequently, Dinic's algorithm forms at most n layered networks and runs in $\mathrm{O}(n^2m)$ time, since as shown below a blocking flow can be computed in $\mathrm{O}(nm)$. Pseudocode of this algorithm can be found in Figure 5.8.

We now explain how to compute blocking flows in the layered network L_f. The *main idea* is to repeatedly use *depth first searches* to find augmenting paths in the layered network. The augmentation of flow along an edge (u, v) reduces the residual capacity of edge (u, v) and increases the residual capacity of the reversal edge (v, u). However, each edge of the layered network is admissible, and therefore Dinic's algorithm does not add reversal edges to the layered network. Consequently, the length of every augmenting path is $d(s)$ and in an augmenting path for every edge (u, v) we have $d(u) = d(v) + 1$, i.e. the level decreases with each edge. These facts allow us to determine an augmenting path in the layered network, on average, in $\mathrm{O}(n)$ time. Pseudocode for the algorithm can be found in Figure 5.9. The

```
Function blockingFlow($H = (V, E)) : E \to \mathbb{N}$
    $p$=$\langle s \rangle$ : Path        $v$=NodeRef : $p$.last()
    $f_b$:= 0
    loop                                                          // Round
        if $v = t$ then                                           // breakthrough
            $\delta$:= $\min\{c(e) - f_b(e) : e \in p\}$
            foreach $e \in p$ do
                $f_b(e) \mathrel{+}= \delta$
                if $f_b(e) = c(e)$ then remove $e$ from $E$
            $p$:= $\langle s \rangle$
        else if $\exists e = (v, w) \in E$ then $p$.pushBack($w$)   // extend
        else if $v = s$ then return $f_b$                         // done
        else delete the last edge from $p$ in $p$ and $E$           // retreat
```

Figure 5.9: Pseudocode for computing a blocking flow. The main idea is to repeatedly use DFS to find augmenting paths.

running time of this algorithm is basically $\#_{extends} + \#_{retreats} + n \cdot \#_{breakthroughs}$. We can have at most m breakthroughs since in each breakthrough at least one edge is saturated. The number of $\#_{retreats}$ is also bounded through m since the edge is deleted from p and E. The number $\#_{extends}$ is lower or equal to $\#_{retreats} + n \cdot \#_{breakthroughs}$ since a retreat cancels 1 extend and a breakthrough cancels $\leq n$ extends. Therefore the overall running time of this algorithm is $\mathrm{O}(m + nm) = \mathrm{O}(nm)$. If we have some further assumptions this time can even get better. For example, if we have unit edge capacities then a breakthrough saturates *all* edges on p, i.e. we have amortized constant cost per edge, yielding a running time of $\mathrm{O}(m + n)$ for this subroutine.

Theorem 4
On a unit network, i.e. unit edge capacities, Dinic's algorithms makes at most $\mathrm{O}(\sqrt{m})$ *rounds.*

Proof.
Consider the layered network after $\sqrt{m}$ iterations. It has at most m edges. The average number of edges between any two layers is $m/\sqrt{m} = \sqrt{m}$. Hence, there must be at least one layer i with at most $\sqrt{m}$ edges. Now consider the cut $S = \{v \mid d(v) < i\}$. It has capacity at most $\sqrt{m}$. Since each blocking flow augmentation increases flow by at least one, after $\sqrt{m}$ additional blocking flow augmentations, a maximum flow has been found. Hence, no more than $2\sqrt{m} = \mathrm{O}(\sqrt{m})$ blocking flow augmentations are needed.

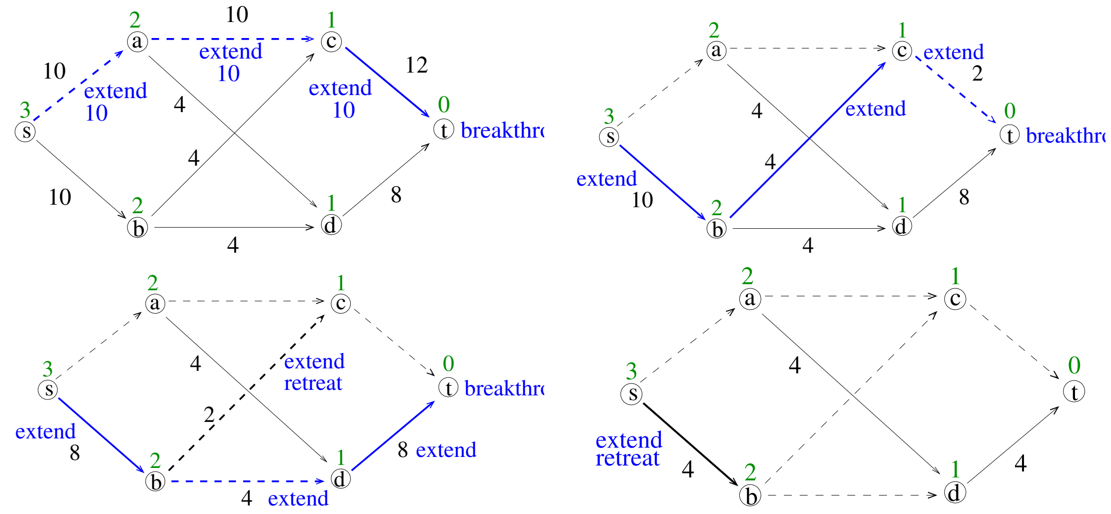


Figure 5.10: An example run of the blocking flow subroutine.

Putting these things together yields a total time for Dinic's algorithm on unit capacity networks of $O((n+m)\sqrt{m})$. A more detailed analysis gives a running time of $O\big(m\min\{(m^{1/2}, n^{2/3}\}\big)$.

Another improvement for general networks was proposed by Sleator and Tarjan in 1983. They developed the dynamic tree data structure and used it to improve the worst-case complexity of Dinic's algorithm from $O(n^2m)$ to $O(mn\log n)$. Dynamic trees are a special type of data structure that permits us to implicitly send flow on paths of length n in $O(\log n)$ steps on average. Since then, researchers have used this data structure on many occasions to improve the performance of a range of network flow algorithms.

5.2.4 Disadvantage of Augmenting Path Algorithms

The inherent drawback of the augmenting path algorithms is the computationally expensive operation of sending flow along a path, which requires $O(n)$ time in the worst case. Preflow-push algorithms do not suffer from this drawback and obtain dramatic improvements in the worst-case complexity. To understand this point better, consider the (artificially extreme) example shown in Figure 5.11.

When applied to this problem, any augmenting path algorithm would discover l augmenting paths, each of length 6, and would augment 1 unit of flow along each of these paths. Observe, however, that although all of these paths share the same first eight 4 edges, each augmentation traverses all of these edges. If we could have

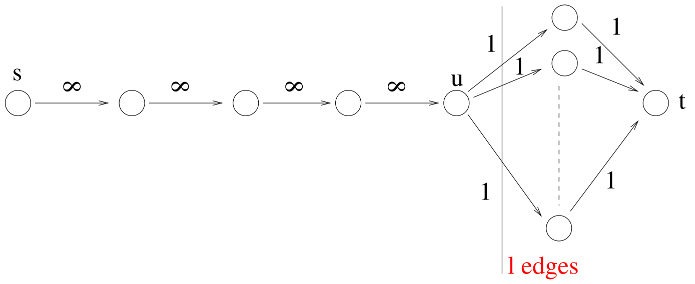


Figure 5.11: Drawback of the augmenting path algorithm.

sent l units of flow from s to node u and then sent 1 unit of flow along l different paths of length 2, we would have saved the repetitive computations in traversing the common set of edges. This is the essential idea underlying the preflow-push algorithms.

5.3 Preflow-Push Algorithms

We now study a class of algorithms, known as *preflow-push* algorithms, for solving the maximum flow problem. These algorithms are more general, more powerful, and more flexible than augmenting path algorithms. The best preflow-push algorithms currently outperform the best augmenting path algorithms in theory as well as in practice. In this section we study the generic preflow-push algorithm.

Augmenting path algorithms send flow by augmenting along a path. The basic operation further decomposes into the more elementary operation of sending flow along individual edges. Thus sending a flow of δ units along a path of k edges decomposes into k basic operations of sending a flow of δ units along each of the edges of the path. We shall refer to each of these basic operations as a push (see Figure 5.12). The preflow-push algorithms push flows on individual edges instead of augmenting paths.

Definition 6
A preflow f is a flow where the flow conservation *constraint is relaxed to*

$$\text{excess}(v) := \overbrace{\sum_{(u,v)\in E} f_{(u,v)}}^{\textit{inflow}} - \overbrace{\sum_{(v,w)\in E} f_{(v,w)}}^{\textit{outflow}} \geq 0 \ .$$

This implies for a preflow that it is allowed that the total inflow exceeds the total outflow. A node v is called active *if and only if* excess$(v) > 0$*. As an exception,*

s and t are never active. The residual network G^f with respect to a preflow f is defined as in the case of a flow.

The preflow-push algorithms maintain a preflow at each intermediate stage. In a preflow, we have $\text{excess}(v) \geq 0$ for each $v \in V \backslash \{s, t\}$. Moreover, because no edge is leaving from t we have $\text{excess}(t) \geq 0$ as well. Therefore node s is the only node with negative excess. The basic operation to manipulate a preflow is a *push*:

```
Procedure push($e = (v, w), \delta$)
    assert $\delta > 0$
    assert residual capacity of $e \geq \delta$
    assert $\text{excess}(v) \geq \delta$
    $\text{excess}(v) -= \delta$
    $\text{excess}(w) += \delta$
    if $e$ is reverse edge then $f(\text{reverse}(e)) -= \delta$
    else $f(e) += \delta$
```

Figure 5.12: The basic push operation.

A push of δ units from node v to w decreases both $\text{excess}(v)$ and $c^f_{(v,w)}$ by δ units and increases both $\text{excess}(w)$ and $c^f_{(w,v)}$. We call a push *saturating* if $\delta = c^f_e$ and *non-saturating* if $\delta < c^f_e$. A non-saturating push at node v reduces $\text{excess}(v)$ to zero. A saturating push across e removes e from the residual network and either kind of push adds e^{rev} to the residual network (if it is not already there).

The question is now which pushes to perform? Goldberg and Tarjan suggested to put nodes of G (and hence G^f) onto layers with t on the bottom-most layer and to perform only pushes which transport excess to a lower layer. We use $d(v)$ to denote the (number of the) layer containing v. We call an edge $e = (v, w) \in G^f$ *eligible* if $d(w) < d(v)$.

Let us summarize: a push across an edge $e = (v, w) \in G^f$ can be performed if v is active and e is eligible. It moves $\delta \leq \min(\text{excess}(v), c^f_e)$ units of flow from v to w. If e is also an edge of G then $f(e)$ is increased by δ, and if e is the reversal of an edge of G then $f(e)$ is decreased by δ.

What are we going to do when v is active but there is no eligible edge out of v? In this situation v is *relabeled* by increasing $d(v)$ by one, i.e. the purpose of the relabel operation is to create at least one edge on which the algorithm can perform further pushes. The generic preflow-push algorithm is proposed in Figure 5.13.

```
Procedure genericPreflowPush(G=(V,E), f)
    forall $e = (s, v) \in E$ do push$(e,$ cap$(e))$                         // saturate
    $d(s)$:= $n$
    $d(v)$:= 0 for all other nodes
    while $\exists v \in V \setminus \{s, t\}$ : excess$(v) > 0$ do              // active node
        if $\exists e = (v, w) \in E_f : d(w) < d(v)$ then                  // eligible edge
            choose some $\delta \leq \min\left\{\text{excess}(v), c_e^f\right\}$
            push$(e, \delta)$                                    // no new steep edges
        else $d(v)$++                                  // relabel. No new steep edges
```

Figure 5.13: The generic preflow-push algorithm.

The obvious choice for δ is $\min\left\{\text{excess}(v), c_e^f\right\}$. We call an edge $e = (v, w) \in G^f$ *steep* if $d(w) < d(v) - 1$, i.e., if it reaches down by two or more levels.

It might be instructive to visualize the generic preflow-push algorithm in terms of a physical network: edges represent flexible water pipes, nodes represent joints and the distance function measures how far nodes are above the ground. In this network we wish to send water from the source to the sink. In addition, we visualize pushed flow as water flowing downhill. Initially, we move the source node upward, and water flows to its neighbors. In general, water flows downhill towards the sink; however, occasionally flow becomes trapped locally at a node that has no downhill neighbors. At this point we move the node upward, and again water flows downhill toward the sink. As we continue to move nodes upward, the remaining excess eventually flows back toward the source. The algorithm terminates when all the water flows either into the sink or flows back to the source.

In the following we show that an *arbitrary* preflow-push algorithm finds a maximum flow in time $\mathrm{O}(n^2m)$. We divide this into two steps. The first step shows that if the algorithm terminates then it computed a maximum flow. Then we show the running time of the algorithm.

Lemma 5

The algorithm maintains a preflow and does not generate steep edges. The nodes s and t stay on levels 0 and n, respectively.

Proof.

The algorithm clearly maintains a preflow. After the initialization, each edge in G^f either connects two nodes on level zero or connects a node on level zero to node s on level n. Thus, there are no steep edges (there are not even any eligible edges). A relabeling of a node v does not create a steep edge since a node is only relabeled if there are no eligible edges out of it. A push across an edge

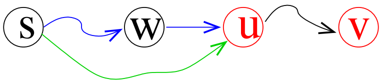
$e = (v, w) \in G^f$ may add the edge $(w, v) \in G^f$. However, this edge is not even eligible. Only active nodes are relabeled and only nodes different from s and t can be active. Thus s and t stay on layers n and 0, respectively. The preceding lemma has an interesting interpretation. Since there are no steep edges, any path from v to t must have length (=number of edges) at least $d(v)$ and any path from v to s must have length at least $d(v) - n$. Thus, $d(v)$ is a lower bound on the distance from v to t and $d(v)-n$ is a lower bound on the distance from v to s.

The next lemma shows that active nodes can always reach s in the residual network (since they must be able to send their excess back to s). It has the important consequence that d-labels are bounded by $2n - 1$.

Lemma 6
If v is active then there is path from v to s in G^f:

$$\forall \textit{ active nodes } v : \textit{excess}(v) > 0 \Rightarrow \exists \textit{ path } \langle v, \ldots, s \rangle \in G_f.$$

Proof.
The following is an equivalent invariant: There is an s–v path with nonzero flow to any active node. Suppose now that $\mathsf{push}((u, w), \delta)$ would destroy this invariant for a path $\langle s, \ldots, w, u, \ldots, v \rangle$. Then the path $\langle s, \ldots, u, \ldots, v \rangle$ reestablishes it.



Lemma 7
No distance label ever reaches $2n$ ($\forall v \in V : d(v) < 2n$).

Proof.
Assume that a node v is moved to level $2n = d(v)$. Since only active nodes are relabeled, the lemma above implies the existence of a path p (and hence simple path) in G^f from a node on level $2n$ to s (which is on level n). This path has at most $n - 1$ nodes. We have $d(s) = n$ and since the algorithm does not generate steep edges $d(v)$ is lower than $2n$ which is a contradiction.

Lemma 8
When genericPreflowPush terminates f is a maximum flow.

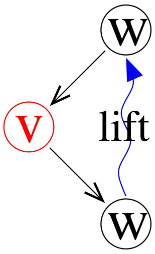
Proof.
When the algorithm terminates, there are no active nodes, i.e. $\forall v \in V \setminus \{s, t\}$: excess$(v) = 0$, and hence the algorithm terminates with a flow. Call it f. In G^f there can be no path from s to t since any such path must contain a steep edge (since s is on level n, t is on level 0). Thus, f is a maximum flow (Max-Flow Min-Cut Theorem).

Lemma 9
There are at most $2n^2$ relabels.

Proof.
No distance label ever exceeds $2n$ ($d(v) \leq 2n$). That means v is relabeled at most $2n$ times. Hence, we have at most $|V| \cdot 2n = 2n^2$ relabel operations.

Lemma 10
There at most nm saturating pushes.

Proof.
A saturating push across an edge $e = (v, w) \in G^f$ removes e from G^f. We claim that v has to be relabeled at least twice before the next push across e and hence there can be at most n saturating pushes across any edge. To see the claim, observe that only a push across e^{rev} can again add e to G^f. Since pushes occur only across eligible edges, w must be relabeled at least twice after the saturating push across e and before the next push across e^{rev}. Similarly, it takes two relabels of v before e becomes eligible again.



It is more difficult to bound the number of non-saturating pushes. It depends heavily on which active node is selected for pushing and on which edge is selected for pushing. Recall that a non-saturating push reduces excess(v) to zero, i.e. deactivates a node.

Lemma 11
The number of non-saturating pushes is $\mathrm{O}(n^2 m)$ if $\delta = \min\left\{\text{excess}(v), c_e^f\right\}$ for arbitrary *node and edge selection rules. (*arbitrary preflow-push*).*

Proof.
The proof makes use of a potential function argument. Consider the potential function

$$\Phi := \sum_{\{v:v \text{ is active}\}} d(v)$$

We will show:

1. $\phi \geq 0$, and $\phi = 0$ initially.
2. A non-saturating push decreases ϕ by at least one.
3. A relabeling increases ϕ by one.
4. A saturating push increases ϕ by at most $2n$.

Suppose that we have shown (1) to (4). By (3) and (4) the total increase of ϕ is at most $2n^2 + nm2n$, since there are at most $2n^2$ relabel operations and at most nm saturating pushes. By (1) the total decrease can be no larger than this. Thus, the number of non-saturating pushes can be at most $\mathrm{O}(n^2m)$.

Operation	$\Delta(\Phi)$	How many times?	Total effect
relabel	1	$\leq 2n^2$	$\leq 2n^2$
saturating push	$\leq 2n$	$\leq nm$	$\leq 2n^2m$
non-saturating push	≤ -1		

It remains to show (1) to (4). (1) is obvious. For (2) we observe that a non-saturating push deactivates a node. It may or may not activate a node at the level below. In either case, ϕ decreases by at least one. For (3) we observe that a relabeling of v increases $d(v)$ by one and for (4) we observe that a saturating push may activate a node and that all distance labels are bounded by $2n$.

Pushing Excess Out of a Node: Let v be a node with positive excess. We want to push flow out of v along eligible edges. An edge $e \in G^f$ is either also an edge of G (and then $f(e) < c_e$) or the reversal of an edge of G (and then $f(e^{\text{rev}}) > 0$)). We therefore iterate over all edges out of v and all edges into v. For each edge out of v we push $\max(\text{excess}(v), c_e - f_e)$. If a push decreases the excess of v to zero we break from the loop. However this implementation yields a degree-factor in the running time which is easily removed by means of the so called *current edge data structure*. We maintain for each node v an edge pointer *currentEdge* to its sequence of outgoing edges in G^f such that all edges preceding this pointer in v adjacency list are not eligible. Recall that an edge (v, w) is eligible if the layer of w is one less than the layer of v and that no edge goes down more that one layer. Thus relabeling w cannot make (v, w) eligible, and reversing an edge in an augmentation cannot make the edge eligible (because all edges in the augmenting path go from lower layers to higher layers after the augmentation). Only relabeling v can make an edge out of v eligible. With these observations it is easy to maintain the invariant that all edges preceding currentEdge in v's adjacency list are not eligible: When v is relabel we reset the currentEdge pointer. This yields the following Lemma.

Lemma 12
Total cost for searching $\leq \sum_{v \in V} 2n \cdot \mathsf{degree}(v) = 4nm = \mathrm{O}(nm)$

Theorem 13
The arbitrary preflow-push algorithm finds a maximum flow in time $\mathrm{O}(n^2m)$.

Proof.
Lemma 8 gives us partial correctness. The time for initialization is bounded through $\mathrm{O}(n+m)$. To maintain a set of active nodes we use for example a stack or a FIFO and we use reverse edge pointers to implement push. Lemma 9 bounds the number of relabel operations to $2n^2$. The saturating pushes are less or equal than nm (Lemma 10) and the non-saturating pushes are in $\mathrm{O}(n^2m)$ (Lemma 11). Lemma 12 states the overall search time for eligible edges is in $\mathrm{O}(nm)$. Summing up yields a total time which is in $\mathrm{O}(n^2m)$. There are multiple ways to improve this running time. The first one is the *FIFO-rule*: The active nodes are kept in a queue and the first node in the queue is always selected. When a node is relabeled or activated the node is added to the rear of the queue. The number of non-saturating pushes is $\mathrm{O}(n^3)$ when the FIFO-rule is uses. This bound is due to Goldberg.

Another variant follows the *Highest Level*-rule: An active node on the highest level, i.e., with the maximal dist-value, is selected. Observe that when a maximal level active node is relabeled, it will be the unique maximal active node after the relabel. Thus, this rule guarantees that when a node is selected, pushes out of the node will be performed until the node becomes inactive. The number of non-saturating pushes is $\mathrm{O}(n^2\sqrt{m})$ when the Highest Level-rule is uses. This rule can be implemented using a bucket priority queue.

Lemma 14
When the Highest Level-rule is used, the number of non-saturating pushes is $\mathrm{O}(n^2\sqrt{m})$.

Proof.
Again we use a potential function argument. Let $K = \sqrt{m}$; this choice of K will become clear at the end of the proof. For a node v, let

$$d'(v) := \frac{|\{w : d(w) \leq d(v)\}|}{K}$$

which basically is the scaled number of dominated nodes. We consider

$$\Phi := \sum_{\{v : v \text{ is active}\}} d'(v)$$

as potential function. We split the execution into phases. We define a phase to consist of all pushes between two consecutive changes of

$$d^* := \max\{d(v) : v \text{ is active}\}$$

and call a phase *expensive* if it contains more than K non-saturating pushes, and *cheap* otherwise. We show:

1. $\leq 4n^2$ phases and $\leq 4n^2K$ non-saturating pushes in all cheap phases together

2. $\Phi \geq 0$ always, $\Phi \leq n^2/K$ initially (obvious)

3. a relabel or saturating push increases Φ by at most n/K.

4. a non-saturating push does not increase Φ.

5. an expensive phase with $Q \geq K$ non-saturating pushes decreases Φ by at least Q.

Suppose that we haven shown (1) to (5). (3) and (4) imply that the total increase of ϕ is at most $(2n^2 + mn)n/K$ and hence the total decrease can be at most this number plus n^2/K by (2). The number of non-saturating pushes in expensive phases is therefore bounded by $(2n^3 + n^2 + mn^2)/K$. Together with (1) we conclude that the total number of non-saturating pushes is at most

$$(2n^3 + n^2 + mn^2)/K + 4n^2K.$$

Observing that $n = \mathrm{O}(m)$ and that the choice $K = \sqrt{m}$ balances the contributions from expensive and cheap phases, we obtain a bound of $\mathrm{O}(n^2\sqrt{m})$.

It remains to prove (1) to (5). For we observe that $d^* = 0$ initially, $d^* \geq 0$ always, and that only a relabel can increase d^*. Thus, d^* is increased at most $2n^2$ times, decreased no more than this, and hence changed at most $4n^2$. Therefore the number of phases is bounded through $4n^2$. This directly yields that the number of non-saturating pushes in all cheap phases together is less or equal than $4n^2K$. (2) is obvious. (3) follows from the observation that $d'(v) \leq n/K$ for all v and at all times. A relabel of v can increase only the d'-value of v. A saturating push on (u, w) may activate only w. For (4) observe that a non-saturating push across an edge (v, u) deactivates v, activates u (if it is not already active), and that $d'(u) \leq d'(v)$ (we do not push flow away from the sink). For (5) consider an expensive phase containing $Q \geq K$ non-saturating pushes. By definition of a phase, d^* is constant during a phase, and hence all Q non-saturating pushes must be out of nodes at level d^*. The phase is finished either because level d^* becomes empty or because a node is moved from level d^* to level $d^* + 1$. In either case, we conclude that level d^* contains $Q \geq K$ nodes at all times during the phase. Thus, each non-saturating push in the phase decreases ϕ by at least one (since $d'(u) \leq d'(v) - 1$ for a push from v to u).

5.3.1 Heuristic Improvements

What is the best case running time of the algorithm? The running time is $\Omega(n^2)$ if $\Omega(n)$ nodes need to be lifted above level n. This is usually the case. The best case behavior of the other parts of the algorithm is $\mathrm{O}(m)$ and hence the cost of relabeling dominates the best case running time. In this section we will describe several heuristics that frequently reduce the time spent in relabeling nodes and as a side-effect reduce the time spent in all other operations. The heuristics will turn the preflow-push algorithm into a highly effective algorithm for solving flow problems.

The Local Relabeling Heuristic: The *local relabeling heuristic* applies whenever a node is relabeled. It increases the dist-value of v to

$$d(v){:=}\,1+\min\left\{d(w):(v,w)\in G_f\right\}.$$

Observe that v is active whenever it is relabeled and that an active node has at least one outgoing edge in G^f. The expression above is therefore well defined. When v is relabeled non of the outgoing edges is eligible and hence $d(w)\geq d(v)$ for all $(v,w)\in G^f$. Thus the local relabeling heuristic increases $d(v)$ by at least one. It may increase it by more than one. The correctness of the heuristic follows from the following alternative description: when a node is relabeled, continue to relabel it until there is an eligible edge out of it.

The Global Relabeling Heuristic: The *global relabeling heuristic* updates the distance values of all nodes. It sets $d(v):=G_f$.reverseBFS(t) for nodes that can reached in G_f (nodes that aren't reachable get another d-value which is not defined here but in the LEDA book of Mehlhorn/Näher). A breadth-first search requires time $\mathrm{O}(m)$. It should therefore not be applied to frequently. We will apply it initially and every $\mathrm{O}(m)$ edge inspections. In this way $\Omega(m)$ time is spent between applications of the global relabel heuristic and hence the worst case running time is increased by at most a constant factor. The best case can improve significantly.

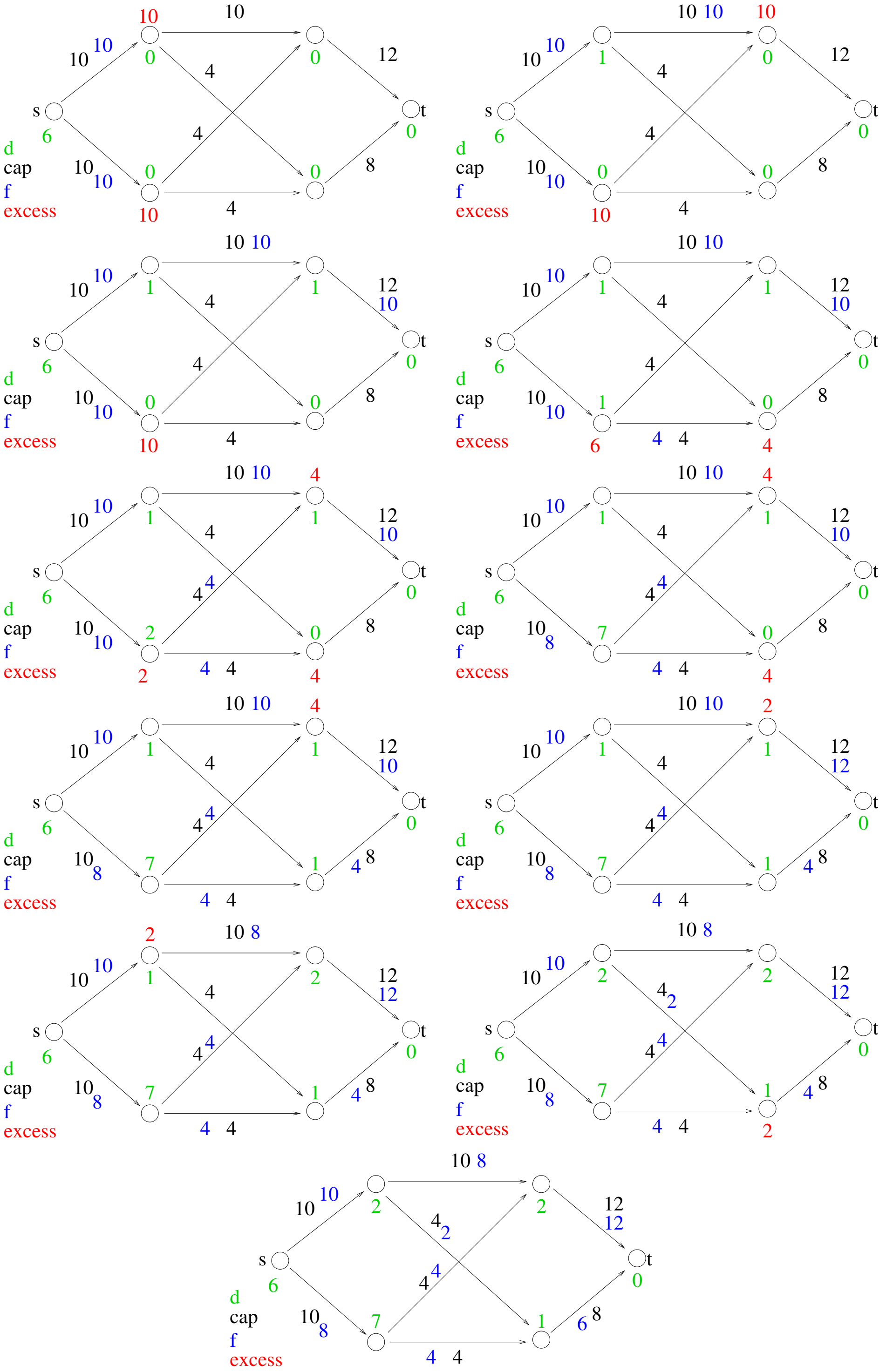


Figure 5.14: An example run of the arbitrary preflow-push algorithm (12 pushes).

The Gap Heuristic: We come to our last optimization. Consider a relabeling of a node v in phase one and let d_v be the layer of v before the relabeling. If the layer d_v becomes empty by the relabeling of v, then v cannot reach t anymore in G^f after the relabeling, since any edge crossing the now empty layer would be steep. If v cannot reach t in G^f then no node reachable from v in G^f can reach t. We may therefore move v and all nodes reachable from v to layer n whenever the old layer of v becomes empty by relabeling of v. This is called the *gap heuristic*.

5.3.2 Experimental Results

In this section we present experimental results for the different algorithms and configurations we have seen. We use four classes of graphs:

- Random: n nodes, $2n + m$ edges; all edges (s, v) and (v, t) exist
- Cherkassky and Goldberg (1997) (two graph classes)
- Ahuja, Magnanti, Orlin (1993)

Timings: Random Graphs

Rule	BASIC	HL	LRH	GRH	GAP	LEDA
FF	5.84	6.02	4.75	0.07	0.07	—
	33.32	33.88	26.63	0.16	0.17	—
HL	6.12	6.3	4.97	0.41	0.11	0.07
	27.03	27.61	22.22	1.14	0.22	0.16
MF	5.36	5.51	4.57	0.06	0.07	—
	26.35	27.16	23.65	0.19	0.16	—

Timings: CG1

Rule	BASIC	HL	LRH	GRH	GAP	LEDA
FF	3.46	3.62	2.87	0.9	1.01	—
	15.44	16.08	12.63	3.64	4.07	—
HL	20.43	20.61	20.51	1.19	1.33	0.8
	192.8	191.5	193.7	4.87	5.34	3.28
MF	3.01	3.16	2.3	0.89	1.01	—
	12.22	12.91	9.52	3.65	4.12	—

Timings: CG2

Rule	BASIC	HL	LRH	GRH	GAP	LEDA
FF	50.06	47.12	37.58	1.76	1.96	—
	239	222.4	177.1	7.18	8	—
HL	42.95	41.5	30.1	0.17	0.14	0.08
	173.9	167.9	120.5	0.36	0.28	0.18
MF	45.34	42.73	37.6	0.94	1.07	—
	198.2	186.8	165.7	4.11	4.55	—

Timings: AMO

Rule	BASIC	HL	LRH	GRH	GAP	LEDA
FF	12.61	13.25	1.17	0.06	0.06	—
	55.74	58.31	5.01	0.1399	0.1301	—
HL	15.14	15.8	1.49	0.13	0.13	0.07
	62.15	65.3	6.99	0.26	0.26	0.14
MF	10.97	11.65	0.04999	0.06	0.06	—
	46.74	49.48	0.1099	0.1301	0.1399	—

FF=FIFO node selection, HL=hightest level, MF=modified FIFO
HL= $d(v) \geq n$ is special,
LRH=local relabeling heuristic, GRH=global relabeling heuristics

Asymptotics, $n \in \{5000, 10000, 20000\}$

Gen	Rule	GRH			GAP			LEDA		
rand	FF	0.16	0.41	1.16	0.15	0.42	1.05	—	—	—
	HL	1.47	4.67	18.81	0.23	0.57	1.38	0.16	0.45	1.09
	MF	0.17	0.36	1.06	0.14	0.37	0.92	—	—	—
CG1	FF	3.6	16.06	69.3	3.62	16.97	71.29	—	—	—
	HL	4.27	20.4	77.5	4.6	20.54	80.99	2.64	12.13	48.52
	MF	3.55	15.97	68.45	3.66	16.5	70.23	—	—	—
CG2	FF	6.8	29.12	125.3	7.04	29.5	127.6	—	—	—
	HL	0.33	0.65	1.36	0.26	0.52	1.05	0.15	0.3	0.63
	MF	3.86	15.96	68.42	3.9	16.14	70.07	—	—	—
AMO	FF	0.12	0.22	0.48	0.11	0.24	0.49	—	—	—
	HL	0.25	0.48	0.99	0.24	0.48	0.99	0.12	0.24	0.52
	MF	0.11	0.24	0.5	0.11	0.24	0.48	—	—	—

5.4 Minimum Cost Flows

The idea of augmenting paths extends to a more general network flow problem. We define $G = (V, E)$, f, excess, and c as for maximum flows. Let cost : $E \to \mathbb{R}$ denote the edge costs per unit of flow in addition to a capacity. Consider further supply : $V \to \mathbb{R}$ with $\sum_{v \in V}$ supply$(v) = 0$. A *negative supply* is called a *demand*. The *cost* of a flow f is cost$(f) := \sum_{e \in E} f(e)$cost$(e)$. A flow is *minimum cost* if among all flows of the same value it has minimum costs. The *minimum cost flow problem* is that of finding a maximum flow of minimum cost. The min-cost flow problem is as follows:

$$\text{minimize } c(f) := \sum_{e \in E} f(e)\text{cost}(e)$$

$$\text{subject to: } \begin{cases} \forall v \in V : \text{excess}(v) = -\text{supply}(v) & \text{// flow conservation constraints} \\ \forall e \in E : f(e) \leq c(e) & \text{// capacity constraints} \end{cases}$$

Cycle Canceling Algorithm for Min-Cost Flow

To propose the algorithm we first need to define *residual cost*: For $e \in E$, we set cost$_f(e)$ = cost(e) (edge exists in G_f iff $f(e) < c(e)$), and we set cost$_f$(reverse(e)) = $-$cost(e) (edge exists in G_f iff $f(e) > 0$).

Lemma 15
A feasible flow is optimal iff $\nexists$ *cycle* $C \in G_f$: *cost*$_f(C) = \sum_{e \in C}$ *cost*$_f(e) < 0$

Proof.
not here (see [3], p. 307)

We now propose the algorithm which has pseudo-polynomial running time:

```
$f$:= any feasible flow
invariant $f$ is feasible
while $\exists$ cycle $C$ : cost$_f(C) < 0$ do
    augment flow around $C$
```

Figure 5.15: The cycle canceling algorithm.

But how do we compute a feasible flow? We can do this using maximum flow algorithms and the following construction: First we set up a maximum flow network G^* starting with the min cost flow problem G:

- Add a vertex s
- $\forall v \in V$ with supply$(v) > 0$, add edge (s, v) with cap. supply(v)
- Add a vertex t
- $\forall v \in V$ with supply$(v) < 0$, add edge (v, t) with cap. $-$supply(v)
- find a *maximum flow* f in G^*

If f saturates the edges leaving s then f is feasible for G. Otherwise there cannot be a feasible flow f' because f' could easily be converted into a flow in G^* with larger value.

Lemma 16 (*Integrality Property*)
If all edge capacities are integral then there exists an integral minimum cost flow.

Proof.
Exercise.
There are better algorithms to solve the problem:

Theorem 17
The min-cost flow problem can be solved in time $\mathrm{O}(mn \log n + m^2 \log \max_{e \in E} c(e))$.

Proof.
For details take the courses in optimization or network flows.

Special Cases of Min Cost Flows

Transportation Problem: The *transportation problem* is a special case of the minimum cost flow problem with the property that the node set V is partitioned into two subsets V_1 and V_2 (of possibly unequal cardinality) so that (1) each node in V_1 is a supply node, (2) each node in V_2 is a demand node, (3) each edge starts in V_1 and ends in V_2. We further have $\forall e \in E : c(e) = \infty$. The classical example of this problem is the distribution of goods from warehouses to customers. In this context the nodes in V_1 represent the warehouses, the nodes in V_2 represent customers (or, more typically, customer zones), and an edge $(u.v)$ represents a distribution channel from warehouse u to customer v.
Minimum Cost Bipartite Perfect Matching: The transportation problem can be used to solve the *Minimum Cost Bipartite Perfect Matching*. We therefore formulate a transportation problem in a bipartite graph $G = (A \cup B, E \subseteq A \times B)$ with $\mathsf{supply}(v) = 1$ for $v \in A$, $\mathsf{supply}(v) = -1$ for $v \in B$. An *integral* flow defines a matching. Reminder: $M \subseteq E$ is a *matching* if (V, M) has maximum degree one.

5.5 Matching Problems

A matching in a graph $G = (V, E)$ is a set of edges with the property that every node is incident to at most one edge in the set; thus a matching induces a pairing of (some of) the nodes in the graph using the edges in E. In a matching, each node is matched with at most one other node, and some nodes might not be matched with any other node. The *matching problem* seeks a matching that optimizes some criteria. Matching problems on bipartite graphs are called *bipartite matching problems*, and those on non-bipartite graphs are called *nonbipartite matching problem* . There are two additional ways of categorizing matching problems: *cardinality matching problems*, which maximize the number of pairs of nodes matched, and *weighted matching problems* which maximize or minimize the weight of the matching (Find a matching M^* such that $w(M^*) := \sum_{e \in M^*} w(e)$ is maximized). Applications of matching problems arise in matching roommates to hostels, matching pilots to compatible airplanes, scheduling airline crews for available flight legs, and assigning duties to bus drivers.

Theorem 18
A maximum weighted matching can be found in time $\mathrm{O}(nm + n^2 \log n)$. *[Gabow 1992]*

Proof.
Not here.

Theorem 19
There is an $\mathrm{O}(m)$ *time algorithm that finds a matching of weight at least* $\max_{\text{matching } M} w(M)/2$. *The algorithm is a* $1/2$*-approximation algorithm [27].*

Proof.
The algorithm is as follows:

```
$M'$:= $\emptyset$
invariant $M'$ is a set of simple paths
while $E \neq \emptyset$ do                                  // find heavy simple paths
    select any $v \in V$ with degree($v$) $> 0$             // select a starting node
    while degree($v$) $> 0$ do                              // extend path greedily
        $(v, w)$:= heaviest edge leaving $v$                               // (*)
        $M'$:= $M' \cup \{(v, w)\}$
        remove $v$ from the graph
        $v$:= $w$
return any matching $M \subseteq M'$ with $w(M) \geq w(M')/2$
//one path at a time, e.g., look at the two ways to take every other edge.
```

We now have to proof the approximation ratio: Let M^* denote a maximum weight matching. It suffices to show that $w(M') \geq w(M^*)$. Therefore we assign each edge to that incident node that is deleted first. Now all e^* in our maximum weight matching M^* are assigned to different nodes (otherwise it wouldn't be a matching). Consider any edge $e^* \in M^*$ and assume it is assigned to node v. Since e^* is assigned to v, it was available in line **(*)**, i.e. it could have been selected in this line. Hence, there is an edge $e \in M'$ assigned to v with $w(e) \geq w(e^*)$.

Lemma 20
There is an $\mathrm{O}\left(m \log \frac{1}{\epsilon}\right)$ *time algorithm that computes a* $\frac{2}{3} - \epsilon$*-approximation of the maximum weight matching problem [69].*

Proof.
The algorithm is as follows.

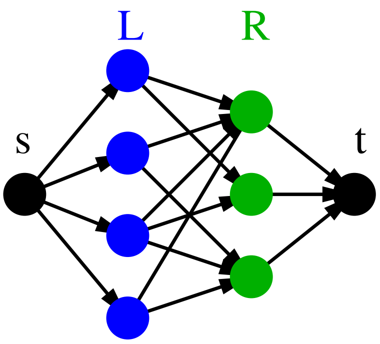
```
$M := \emptyset$
**repeat** $\Theta\left(n \log \frac{1}{\epsilon}\right)$ times:
    Let $X \in V(G)$ be selected uniformly at random
    $M := M \oplus \mathsf{aug}(X)$
**return** $M$
```

For more details look into the paper.

5.5.1 Maximum Cardinality Bipartite Matching

As introduced in the beginning of this Section a maximum cardinality matching is a matching of maximum size. We now consider the problem of finding a maximum cardinality matching in a given bipartite graph $G = (L \cup R, E)$. It turns out that this problem can be reduced to a unit network max-flow problem. The construction is as follows:

$$G' = (\{s\} \cup L \cup R \cup \{t\}, \{(s,u) : u \in L\} \cup E_{L\to R} \cup \{(v,t) : v \in R\}).$$



Here $E_{L \to R}$ are all edges E directed from L to R. The capacity of every edge is set to one. First notice that if G has a matching of size k then G' has a flow of value k. To see this let M be a matching of size k. Now we construct a flow that sends one unit along each edge in M. This flow has value k.

On the other hand if G' has a flow of value k then G has a matching of size k. To prove this consider a flow f of value k. First observe that f is an integral flow. Thus each edge has flow 1 or 0. Consider the set M of edges from L to R that have flow 1. M has k edges because the value of the flow is equal to the number of non-zero flow edges crossing the cut $(L \cup \{s\}, R \cup \{t\})$. Furthermore, each vertex has at most one edge in M incident upon it (since all edges have capacity 1).

Thus, to find a maximum cardinality matching in G, we construct G', find the maximum flow in G' and construct the matching M as above. To solve the unit network max-flow problem G' we can use Dinitz algorithm which yields a $\mathrm{O}((n+m)\sqrt{n})$ algorithm for the maximum cardinality bipartite matching problem.

5.6 Book References

This chapter is largely based on Mehlhorn et. al. [61] as well as Ahuja et. al. [3].

Chapter 6

Randomized Algorithms

Algorithms that make random choices during their execution are called *randomized algorithms*. In contrast, algorithms *not* using randomization are called *deterministic*. Formally, randomized algorithms can be described using a deterministic RAM (random access model) with an additional instruction as machine model. The instruction $R_i := randInt(C)$ assigns a *random* integer between 0 and $C - 1$ to R_i and runs in one time unit.

Note that true randomness cannot be generated by a deterministic machine like our current consumer machines. Nevertheless, there exists a multitude of *pseudo-random generators* producing random numbers that even suffice for cryptographic purposes (e.g. B.B.S. [10], Mersenne twister [52, 53]). Furthermore, dedicated hardware can be applied to obtain true random numbers with the use of physical phenomena (e.g. quantum effects, thermal noise, clock drift).

One or more properties of a randomized algorithm have to be described with *random variables*. Usually, the running time, the error probability or both are said random variables. Depending on which, we classify randomized algorithms in two main varieties:

Monte Carlo A *Monte Carlo* algorithm has a deterministic asymptotic runtime. But it can produce an incorrect result with a small probability p. Repeating the algorithm k times, the probability of failure can be quickly decreased down to p^k. This technique is called *probability boosting*.

Las Vegas A *Las Vegas* algorithm always produces a correct result. But the asymptotic runtime of the algorithm is a random variable. Examples for this kind of algorithms include *quicksort* and *hashing*.

A *Monte Carlo* algorithm can always be transformed into a *Las Vegas* algorithm by repeated executions until a correct result is achieved. The efficiency of this procedure depends on time required to verify the correctness of a result.

This course gives an overview of several interesting randomized algorithms. It is not intended to be a formal introduction into the field of randomized algorithms. The course “Randomisierte Algorithmen” by Thomas Worsch hold in winter term provides a more thorough overview of this subject.

6.1 Sorting

Given a sorting algorithm, it is easy to quickly verify whether it produces a sorted output. It is more complicated and time consuming to verify whether the output is a permutation of the input. Randomized algorithms help us to check the latter property in linear time with high probability.

We say that a sequence $s = \langle e_1, \ldots, e_n \rangle$ is a permutation of $s' = \langle e'_1, \ldots, e'_n \rangle$ if and only if the polynomial

$$q(z) := \prod_{i=1}^{n} (z - \text{field}(\text{key}(e_i))) - \prod_{i=1}^{n} (z - \text{field}(\text{key}(e'_i)))$$

is identically zero (here, field : Key $\to \mathbb{F}$ denotes an injective mapping of the key domain of the elements Key to a field $\mathbb{F}$). Evidently, the polynomial is identically zero if s is a permutation of s' as both products consist of the same linear terms. Otherwise, there exist $m \geq 1$ linear terms in each product that that do not occur in the other. Since they can never be identically equal, the polynomial cannot be identically zero if s is not a permutation of s'.

Note that the polynomial has maximum degree n. Thus, it is either identically zero, or it has at most n zeros. If it is not identically zero and it is evaluated at a random position $z \in \mathbb{F}$, the probability to obtain a zero is at most

$$\mathbb{P}\left[q \neq 0 \land q(x) = 0\right] \leq \frac{n}{|\mathbb{F}|}.$$

With this knowledge we can formulate a randomized algorithm to verify whether s is a permutation of s': First, construct polynomial $q(\cdot)$. Then, evaluate $q(\cdot)$ at a random position $z \in \mathbb{F}$. If the result equals zero, s is a permutation of s' with a probability of $1 - \frac{n}{|\mathbb{F}|}$. The algorithm runs in linear time as a polynomial can be evaluated in linear time. The proposed algorithm is a *Monte Carlo* algorithm.

Note that the choice of field $\mathbb{F}$ and of the injective function field$(\cdot)$ are still open and have to be considered by an actual implementation. As exercise, consider what would be a good choice for both of them.

6.2 Hashing

Let us first recapitulate the general notion of *hashing* before delving into more complex hashing methods. Hashing was first described by Dumey [28] and emerged in the 1950s as a space efficient heuristic for fast retrieval of keys in sparse tables. A survey of the most important classical hashing methods is given by Knuth in [48].

Hash tables are an implementation of the abstract data structure *associative array*, or *dictionary*. An associative array S stores a set of elements and supports the operations *insert*, *remove* and *lookup*. Usually, each element e has an associated unique key $key(e) \in Key$ by which it is identified. We omit this indirection below for clarity of description.

The basic hash table implementation of an associative arrays is simple. We use a *hash function* h to map our elements to a small range $0..m-1$ of integers. These values are used as indices into a *hash table* T of size m that stores the values. Ideally, m should be close to the number of elements we want to store $n = |S|$, so that we waste only little space.

If our hash function h maps distinct elements to the same table entry, we say that the elements *collide*. The strategy by which collisions are resolved or avoided is subject of the individual hashing methods. *Hashing with chaining*, *linear probing* and *double hashing* are some of the most common methods. These methods guarantee *expected* constant time for the three supported operations and require $\mathrm{O}(n)$ space.

6.2.1 Perfect Hashing

As the name suggests, *perfect hashing* allows for hashing without collisions. This property lets perfect hashing guarantee *worst-case* constant time for *lookup* and *remove* in contrast to the previously mentioned methods that could only guarantee *expected* constant time. To avoid collisions, an *injective* hash function h is required. Evaluation of h must be possible in constant time to guarantee constant time for *lookup* and *remove*. In addition, fast construction of the hash function and small representation are desirable. Perfect hashing is explained in more detail in Chapter 4.5 of [62].

Space Requirements. Regarding hashing methods, we have to consider two notions of space requirements. First, we are concerned about the space required to store n elements of a universe of size u. Then, there is the space needed to represent a hash function that maps n elements to m positions in a hash table.

The information theoretical minimum for storing a subset of n elements of a universe of size u is $\lceil \log \binom{u}{n} \rceil$ bits as there are $\binom{u}{n}$ distinct subsets of size n. This constitutes the lower bound for representing a hash table *without associated* *information*. For all practical purposes, this limit has been reached in the previous years. In 1999, Brodnik and Munro [14] presented a solution that only required $(1 + \mathrm{o}(1))$ times the minimum amount of space and offered constant evaluation time. Three years later, Raman et al. [75] reached similar results with smaller $\mathrm{o}(\cdot)$ factors and added functionality in the form of constant time *rank* and *select* operations. Note that both approaches cannot be easily extended to work with additional information associated to each element.

Representing a simple hash function like $h(x) = a \cdot x + c \pmod n$ requires only two integer values. Perfect hash functions however need much more space. In particular, it was shown that the information theoretical minimum is about $1.44n^2/m$ bits. Recently, Botelho et al. presented a linear time algorithm for constructing such a hash function that comes within a factor of two of this bound and that can be evaluated in constant time. Previous near space-optimal approaches either fulfilled this optimality only asymptotically for very large n or required exponential construction time. The construction procedure can be found in [12].

Fast Space Efficient Hashing. In general we would like to store n elements with associated information using little extra space, i.e. $(1+\varepsilon)n$ times the space required for one element. In addition, all important operations (*lookup*, *insert*, *remove*) should run efficiently, preferably in constant time. Considering this scenario, the expected time of a *lookup* operation is $\mathbf{E}[T_{lookup}] \approx \frac{1}{2\varepsilon^2}$ with *linear probing* [20]. The lower bound however was shown to be as small as $\mathbf{E}[T_{lookup}] \approx \log \frac{1}{\varepsilon}$. It is reached by several hashing methods, e.g. *uniform hashing* [90]. This approach uses a hash function that yields not only one position in the hash table but a permutation of all m positions. These positions are tested sequentially to find the first available place for insertion or for locating an element.

Note that analyses of hashing methods usually assume that fully random hash functions are available for free. At first, it is not evident that this assumption is always true. But Dietzfelbinger and Weidling show it to be reasonable by presenting a practical workaround to this problem in [26].

Dynamic Perfect Hashing. Until now, we have only considered *static hashing*, i.e. we know in advance how many elements we want to store. In the dynamic case, this information is not available and a hash table needs to shrink or grow as necessary. Naturally, these operations should take little time or happen rarely.

Dynamic perfect hashing has been analyzed by Dietzfelbinger et al. in [25]. They propose an approach that requires space proportional to the number of elements stored. *Lookup* operations can be performed in *worst-case* constant time and *insert* and *remove* operations take *amortized* constant time using a randomized algorithm. Unfortunately, the constant factor is not small.

6.3 Cuckoo Hashing

An interesting hashing method called *cuckoo hashing* was introduced by Pagh and Rodler in 2001 [68]. A very good introduction aimed at undergraduate students with little algorithmic background was given by Pagh in [72].

Basic *cuckoo hashing* uses a hash table of size $2(1+\varepsilon)n$ to store n elements, i.e. it has a space overhead of factor 2. Two hash functions $h_1(\cdot)$ and $h_2(\cdot)$ are applied to store an element. Truely random hash functions with constant evaluation time are assumed. *Lookup* operations simply check the hash table in both positions provided by the hash function and, thus, work in worst-case constant time. Removal of elements works similarly. The *insert* operation is more interesting: If element x is to be inserted and not already in the hash table, position $h_1(x)$ is tested. If this position is already occupied by element y, x replaces y (just like a European cuckoo swaps his egg with another one in a nest). Then, y is inserted at $h_2(y)$. If this position is also occupied, the procedure repeats. As this approach might loop infinitely, it is terminated after $MaxLoop$ iterations and the hash table is rehashed with two new hash functions before the currently free element is inserted again. The expected insertion time is constant without rehashing. Rehashing occurs with probability $\mathbb{P}\left[\text{rehash necessary}\right] = 1/n$ and takes $\mathrm{O}(n)$ time. Thus, the amortized insertion time is $\mathrm{O}(1)$. Pseudocode for the three operations *lookup*, *insert* and *remove* is presented below.

Note that the original work of Pagh and Rodler assumes two distinct hash tables T_1 and T_2 of size $(1+\varepsilon)n$, one for each hash function. This setup offers the same general properties as the one we described here.

```
Procedure lookup($x$)
    return $T[h_1(x)] == x \vee T[h_2(x)] == x$
```

```
Procedure insert($x$)
    if lookup($x$) then return
    loop $n$ times
        if $T[h_1(x)] == empty$ then
            $T_1[h_1(x)] = x$
            return
        swap($x$, $T[h_1(x)]$)
        if $T[h_2(x)] == empty$ then
            $T_2[h_2(x)] = x$
            return
        swap($x$, $T[h_2(x)]$)
    rehash()
    insert(x)
```

```
Procedure remove($x$)
    if $T[h_1(x)] = x$ then
        $T[h_1(x)] = empty$
        return
    if $T[h_2(x)] = x$ then
        $T[h_2(x)] = empty$
        return
```

The stated runtime guarantees of *insert* remain to be proven. We refer to [68] and [72] for an extensive proof. Here, we only show how to detect if a rebuild is needed. For this purpose, we represent the state of the hash table with a graph model. In this *cuckoo graph* a node corresponds to a position in the hash table. Edges denote elements stored in the hash table. There is an edge between to two nodes $(h_1(x), h_2(x))$ if element x is stored in the hash table. The edge is directed outgoing from the position currently occupied by x. Figure 6.1 depicts a sample cuckoo graph.

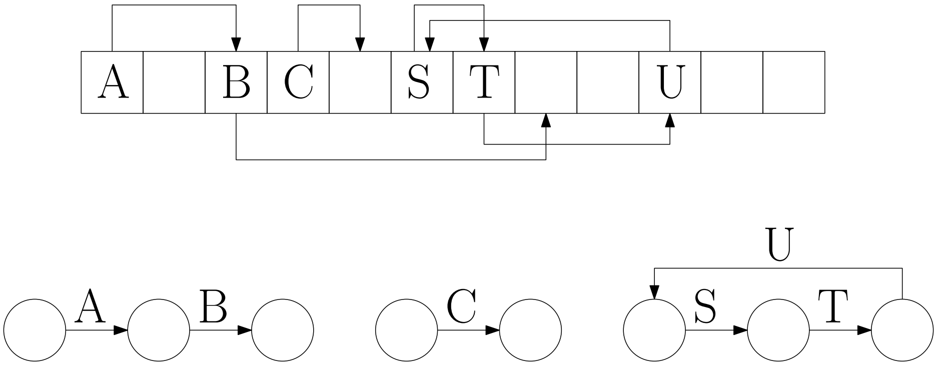


Figure 6.1: The upper image depicts the hash table enriched by edges showing the alternate position of each element. The lower image gives the actual cuckoo graph representation. Each position in the hash table corresponds to a node in the cuckoo graph. Table positions with no edges are omitted for clarity. Directed edges denote stored elements and the direction shows the position to which the element moves if tossed out. Inserting a new element in the position currently occupied by S would not succeed, as S is part of a cycle (together with T, U). The new element would be kicked out again. Inserting a new element in the position of A would work. A would move to the position of B and B to the vacant position next to T.

Lemma 21
Inserting a new element x requires a rehash iff the component containing $h_1(x)$ and $h_2(x)$ in the cuckoo graph contains more edges than nodes.

Proof.
Trying to insert a new element x generates an edge $(h_1(x), h_2(x))$ in the cuckoo graph. Let this edge have an arbitrary orientation. Following, C denotes the component containing $h_1(x)$ and $h_2(x)$.
if-part. If there are more edges than nodes in C, there always exists at least one node with two outgoing edges. This signifies that two elements are stored in the same table position. This is not possible in our hash table. Thus, a rehash is required. Note that switching elements to their alternate position only flips the corresponding edge, retaining the same problem as above.
only-if-part. If there are no more edges than nodes in C, there always exist a configuration of edge orientations so that no node has two outgoing edges. This represents a viable occupation of the hash table. Thus, no rehash is required. □

Taking results from *random graph theory* (see Section 6.4 for a brief introduction), we can quantify the probability of a graph configuration as in Lemma 21. This leads to the afore mentioned probability $\mathbb{P}\left[\text{rehash necessary}\right] = 1/n$ for requiring to rehash the table after inserting a new element.

6.3.1 Space Efficient Cuckoo Hashing

Since its introduction in 2001, there have been made several improvements to cuckoo hasing. Here, we focus on two contributions that extend the idea of cuckoo hashing and reduce its space requirements. The work of Fotakis et al. [31] discusses the effect of using $d > 2$ hash functions (*d-ary cuckoo hashing*), while Dietzfelbinger and Weidling [26] allow d elements to be stored as a block in each table entry (*blocked cuckoo hashing*). Both approaches effectively halve the required space down to a table size of $(1 + \varepsilon)n$ entries for n nodes. They still support *lookup* and *remove* in worst-case constant time $\mathrm{O}(d)$ with $d = \mathrm{O}\big(\log \frac{1}{\varepsilon}\big)$ and insertion in expected constant time $\mathrm{O}\big(\frac{1}{\varepsilon}\big)$.

In the event of a collision during an *insert* operation, basic cuckoo hashing only has one choice in the beginning when element e is first inserted. One can switch e with the element at either position $h_1(e)$ or $h_2(e)$. Subsequent switches are determined by this initial choice. Contrary to this, both new approaches have multiple choices to resolve collisions in each step. Two obvious strategies to handle these options are *random walk* and *BFS* (breath first search). The first strategy randomly switches the to be inserted element with one of the already inserted and conflicting elements. Then it tries to insert this new element. If not succesful, the process is repeated for a maximum number of steps The second strategy systematically explores all possibilities for switching elements. This approach terminates if the search space grows too large. In general, it switches less elements but requires more bookkeeping.

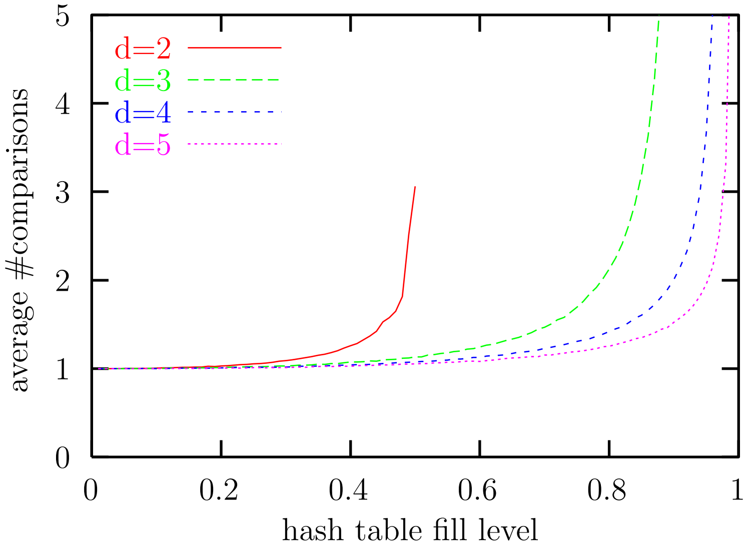


Figure 6.2: Average number of comparisons of an *insert* operation using d*-ary cuckoo hashing* depending on fill level of the the hash table.

Both approaches offer the same asymptotic runtimes but *blocked cuckoo hashing* turned out to be the faster approach in implementation. This can be attributed to its more *cache-efficient* memory access pattern. While d*-ary cuckoo hashing* requires up to d random memory accesses for each element, *blocked cuckoo hashing* requires at most two – at least if all elements in one table entry fit into a single cache line.

We complete the subject with a discussion of the cost of an *insert* operation using d*-ary cuckoo hashing*. Shown in Figure 6.2 is the average number of comparisons when inserting a new element into a hash table at different fill levels. We see that for sparsely populated hash tables the first chosen position can almost always be directly taken. For basic or 2*-ary cuckoo hashing* the required number of comparisons grows quickly with increasing values of d and a sharp threshold at a 50% filled hash table. Thus, on average an *insert* operation will not succeed on an at least half-filled hash table. This implies that a 2-ary cuckoo hash table requires twice as much space as needed for the elements. Fortunately, the threshold quickly approaches 100% for increasing values of d. We conclude that using at least $d > 3$ hash functions yields short insertion times with little space overhead.

6.4 Random Graph Theory

A *random graph* is a graph that is generated by some random process. In 1959, they were first formally defined and analyzed by Erdos and Renyi [29] and independently by Gilbert [33]. Following, we consider the *Erdos Renyi model* by its namesakes for generating random graphs.

Let $\mathcal{G}(n, m)$ be the set of all topological different graphs with n nodes and m edges. Choosing a graph G at random from this set is equal to constructing a graph with n nodes by placing m edges between them at random with equal probability, independently of the other edges. Random graphs that follow this construction model have certain properties with high probability [1]. Particularly interesting is the evolution of connected component sizes with increasing number of edges m:

- $m < (1 - \varepsilon)n/2$: trees and pseudotrees of size $\mathrm{O}(\log n)$
- $m > (1 + \varepsilon)n/2$: one large component of size $\Theta(n)$
- $m > (1 + \varepsilon)n \ln n/2$: one single component

We can see that there is a fundamental state transition at $m \approx n/2$. Below this threshold, only small components exist w.h.p. whereas above this value almost the entire graph is connected w.h.p.. This phenomenon is called *sudden emergence*.

6.5 Book References

This chapter uses excerpts of Mehlhorn et. al. [62] (Chapters 2.8, 4.5) and Motwani et. al. [66] (Chapter 1.2) as well as the referenced papers.

[1] henceforth w.h.p., and in this context meaning “with probabilitty greater than $1 - \mathrm{O}\left(\frac{1}{n}\right)$.”

Chapter 7

External Algorithms

This chapter is mainly based on lecture notes on “Algorithm Engineering“ by Felix Putze and Peter Sanders.

7.1 Introduction

Massive data sets arise naturally in many domains. Spatial data bases of geographic information systems like GoogleEarth and NASA’s World Wind store terabytes of geographically-referenced information that includes the whole Earth. In computer graphics one has to visualize huge scenes using only a conventional workstation with limited memory. Billing systems of telecommunication companies evaluate terabytes of phone call log files. One is interested in analyzing huge network instances like a web graph or a phone call graph. Search engines like Google and Yahoo provide fast text search in their data bases indexing billions of web pages. A precise simulation of the Earth’s climate needs to manipulate with petabytes of data. These examples are only a sample of numerous applications which have to process huge amount of data.

For economical reasons, it is not feasible to build all of the computer’s memory of the fastest type or to extend the fast memory to dimensions that could hold all relevant data. Instead, modern computer architectures contain a memory hierarchy of increasing size, decreasing speed and costs from top to bottom: On top, we have the registers integrated in the CPU, a number of caches, main memory and finally disks, which are often referenced as *external memory* as opposed to *internal memory*.

The internal memories of computers can keep only a small fraction of the large data sets. During processing applications need to access the external memory (e. g. hard disks) very frequently. One such access can be about 10^6 times slower than a main memory access. Therefore, disk accesses (I/Os) become the main bottleneck.

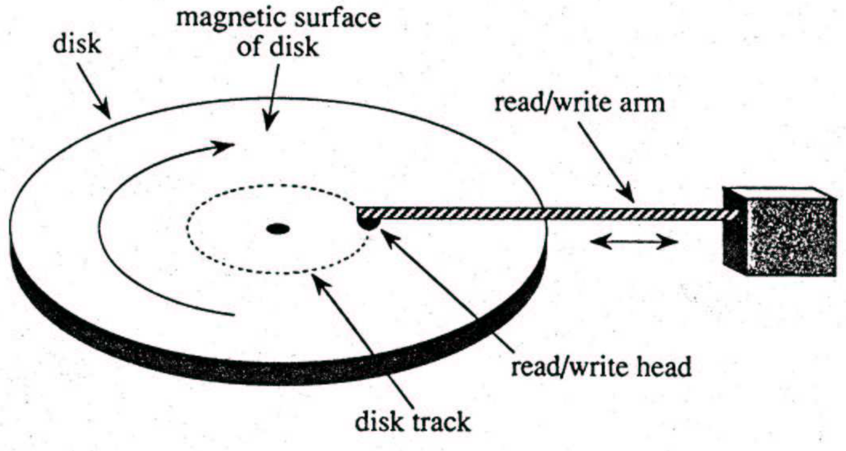


Figure 7.1: schematic construction of a hard disk

The reason for this high latency is the mechanical nature of the disk access. Figure 7.1 shows the schematic construction of a hard disk. The time needed for finding the data position on the disk is called seek time or (seek) latency and averages to about 3-10 ms for modern disks. The seek time depends on the surface data density and the rotational speed and can hardly be reduced because of the mechanical nature of hard disk technology, which still remains the best way to store massive amounts of data. Note that after finding the required position on the surface, the data can be transferred at a higher speed which is limited only by the surface data density and the bandwidth of the interface connecting CPU and hard disk. This speed is called sustained throughput and achieves up to 80 MByte/s nowadays. In order to amortize the high seek latency one reads or writes the data in chunks (blocks). The block size is balanced when the seek latency is a fraction of the sustained transfer time for the block. Good results show blocks containing a full track. For older low density disks of the early 90's the track capacities were about 16-64 KB. Nowadays, disk tracks have a capacity of several megabytes.

Operating systems implement the so called virtual memory mechanism that provides an additional working space for an application, mapping an external memory file (page file) to virtual main memory addresses. This idea supports the Random Access Machine model in which a program has an infinitely large main memory with uniform random access cost. Since the memory view is unified in operating systems supporting virtual memory, the application does not know where its working space and program code are located: in the main memory or (partially) swapped out to the page file. For many applications and algorithms with non-linear access pattern, these remedies are not useful and even counterproductive: the swap file is accessed very frequently; the data code can be swapped out in favor of data blocks; the swap file is highly fragmented and thus many random input/output operations (I/Os) are needed even for scanning.

7.2 The external memory model

If we bypass the virtual memory mechanism, we cannot apply the RAM model for analysis anymore since we now have to explicitly handle different levels of memory hierarchy, while the RAM model uses one large, uniform memory.

Several simple models have been introduced for designing I/O-efficient algorithms and data structures (also called external memory algorithms and data structures). The most popular and realistic model is the Parallel Disk Model (PDM) of Vitter and Shriver, see Figure 7.2. In this model, I/Os are handled explicitly by the application. An I/O operation transfers a block of B consecutive bytes from/to a disk to amortize the latency. The application tries to transfer D blocks between the main memory of size M bytes and D independent disks in one I/O step to improve bandwidth. The input size is N bytes which is (much) larger than M. The main complexity metrics of an I/O-efficient algorithm in this model are:

- I/O complexity: the number of I/O steps should be minimized (the main metric),
- CPU work complexity: the number of operations executed by the CPU should be minimized as well.

The PDM model has become the standard theoretical model for designing and analyzing I/O-efficient algorithms.

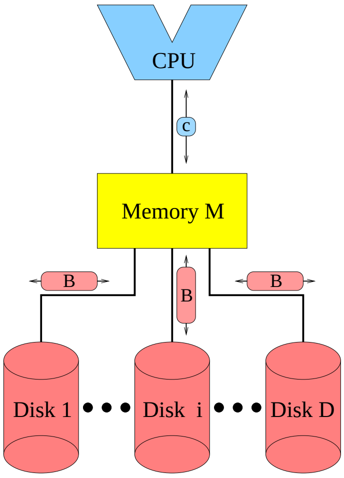


Figure 7.2: Vitter's I/O model with several independent disks

There are some "golden rules" that can guide the process of designing I/O efficient algorithms: Unstructured memory access is often very expensive as it comes with 1 I/O per operation whereas we want $1/B$ I/Os for an efficient algorithm. Instead, we want to *scan* the external memory, always loading the next due block of size B in one step and processing it internally. An optimal scan will only cost $\mathrm{scan}(N) := \Theta(\frac{N}{D \cdot B})$ I/Os. If the data is not stored in a way that allows linear scanning, we can often use *sorting* to reorder and than scan it. As we will see in section 7.4, external sorting can be implemented with $\mathrm{sort}(N) := \Theta(\frac{N}{D \cdot B} \cdot \log_{M/B} \frac{N}{B})$ I/Os.

A simple example of this technique is the following task: We want to reorder the elements in an array A into an array B using a given "rank" stored in array C. This should be done in an I/O efficient way.

```
int[1..N] A,B,C;
for i=1 to N do A[i]:=B[C[i]];
```

The literal implementation would have worst case costs of $\Omega(N)$ I/Os. For $N = 10^6$, this would take $\approx T = 10000$ seconds $\approx$ 3 hours. Using the sort-and-scan technique, we can lower this to $\mathrm{sort}(N)$ and the algorithm would finish in less than a second:

```
SCAN C:     (C[1]=17,1),  (C[2]=5,2),   ...
SORT(1st): (C[73]=1,73), (C[12]=2,12), ...
par SCAN : (B[1],73),    (B[2],12),    ...
SORT(2nd): (B[C[1]],1),  (B[C[2]],2),  ...
```

7.3 Stacks and Queues

The contents of this chapter is from the book [64]

Stacks and queues represent dynamic sets of data elements, and support operations for adding and removing elements. They differ in the way elements are removed. In a *stack*, a remove operation deletes and returns the set element most recently inserted (last-in-first-out), whereas in a *queue* it deletes and returns the set element that was first inserted (first-in-first-out).

Recall that both stacks and queues for sets of size at most N can be implemented efficiently in internal memory using an array of length N and a few pointers. Using this implementation on external memory gives a data structure that, in the worst case, uses one I/O per insert and delete operation. However, since we can read or write B elements in one I/O, we could hope to do considerably better. Indeed this is possible, using the well-known technique of a buffer.

An External Stack

In the case of a stack, the buffer is just an internal memory array of $2B$ elements that at any time contains the k set elements most recently inserted, where $k \leq 2B$. Remove operations can now be implemented using no I/Os, except for the case where the buffer has run empty. In this case a single I/O is used to retrieve the block of B elements most recently written to external memory.

One way of looking at this is that external memory is used to implement a stack with *blocks* as data elements. In other words: The "macroscopic view" in external memory is the same as the "microscopic view" in internal memory.

Returning to external stacks, the above means that at least B remove operations are made for each I/O reading a block. Insertions use no I/Os except when the buffer runs full. In this case a single I/O is used to write the B least recent elements to a block in external memory. Summing up, both insertions and deletions are done in $1/B$ I/O, in the amortized sense. This is the best performance we could hope for when storing or retrieving a sequence of data items much larger than internal memory, since no more that B items can be read or written in one I/O. A desired goal in many external memory data structures is that when reporting a sequence of elements, only $O(1/B)$ I/O is used per element.

An External Queue

To implement an efficient queue we use two buffers of size B, a read buffer and a write buffer. Remove operations work on the read buffer until it is empty, in which case the least recently written external memory block is read into the buffer. (If there are no elements in external memory, the contents of the write buffer is transfered to the read buffer.) Insertions are done to the write buffer which when full is written to external memory. Similar to before, we get at most $1/B$ I/O per operation.

7.4 Multiway Merge Sort

Multiway Merge Sort is covered in [23].

We will now study another algorithm based on the concept of Merge Sort which is especially well suited for external sorting. For external algorithms, an efficient sorting subroutine is even more important than for main memory algorithms because one often tries to avoid random disk accesses by ordering the data, allowing a sequential scan.

Multiway Merge Sort first splits the data into $\lceil n/M \rceil$ *runs* which fit into main memory where they are sorted, see Figure 7.3. We merge these runs until only one is left. Instead of ordinary 2-way-merging, we merge $k := M/B$ runs in a single pass resulting in a smaller number of merge phases. We only have to keep one block (containing the currently smallest elements) per run in main memory. We maintain a priority queue containing the smallest elements of each run in the current merging step to efficiently keep track of the overall smallest element. Figure 7.4 exemplifies a 4-way merging.

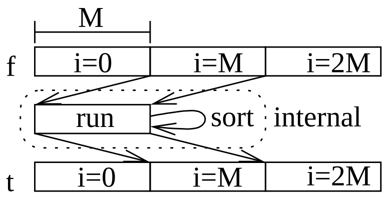


Figure 7.3: Run formation

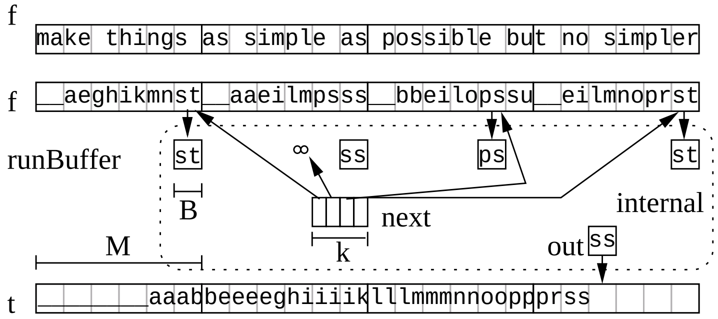

f
make things as simple as possible but no simpler
f
__aeghikmnst__aaeilmpsss__bbeilopssu__eilmnoprst
runBuffer
st
∞
ss
ps
st
B
next
internal
M
k
out
ss
t
aaabbeeeeghiiiiklllmmmnnoopprss

Figure 7.4: Example of 4-way merging with $M = 12$, $B = 2$

Every element is read/written twice for forming the runs (in blocks of size B) and twice for every merging phase. Access granularity is blocks. This leads to the following (asymptotically optimal) total number of I/Os:

$$\frac{2n}{B}\left(1 + \lceil \log_k \#runs \rceil\right) = \frac{2n}{B}\left(1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil\right) := \mathrm{sort}(n) \qquad (7.1)$$

Let us consider the following realistic parameters: $B = 2\mathrm{MB}$, $M = 1\mathrm{GB}$. For inputs up to a size of $n = 512\mathrm{GB}$, we get only one merging phase! In general, this is the case if we can store $\lceil n/M \rceil$ buffers (one for each run) of size B in internal memory (i.e., $n \le M^2/B$). Therefore, only one additional level can increase the I/O volume by 50%.

7.5 Internal work of Multiway Mergesort

Until now we have only regarded the number of I/Os. In fact, when running with several disks our sorting algorithm can very well be compute bound, i.e. prefetching D new blocks requires less time than merging them. We use the technique of *overlapping* to minimize wait time for whichever task is bounding our algorithm in a given environment. Take the following example on run formation (i denotes a run):

Thread A: Loop $\{$ wait-*read* i; *sort* i; post-*write* $i\}$;

Thread B: Loop $\{$ wait-*write* i; post-*read* i+$2\}$;

During initialization, runs 1 and 2 are read, i is set to 1. Thread A sorts runs in memory and writes them to disk. Thread B will wait until run i is finished (and thread A works on $i + 1$) and reads the next run $i + 2$ into the freed space. The thread doing the more intense work will never wait for the other one.

A similar result can be achieved during the merging step: We have an I/O-thread (more, if $D > 1$) and a merging-thread. Merging is I/O bound if and only if $y > DB/2$, where y is the number of elements merged during one I/O step[1] and DB/2 is the number of elements that can be moved in and out the merger during one I/O step.

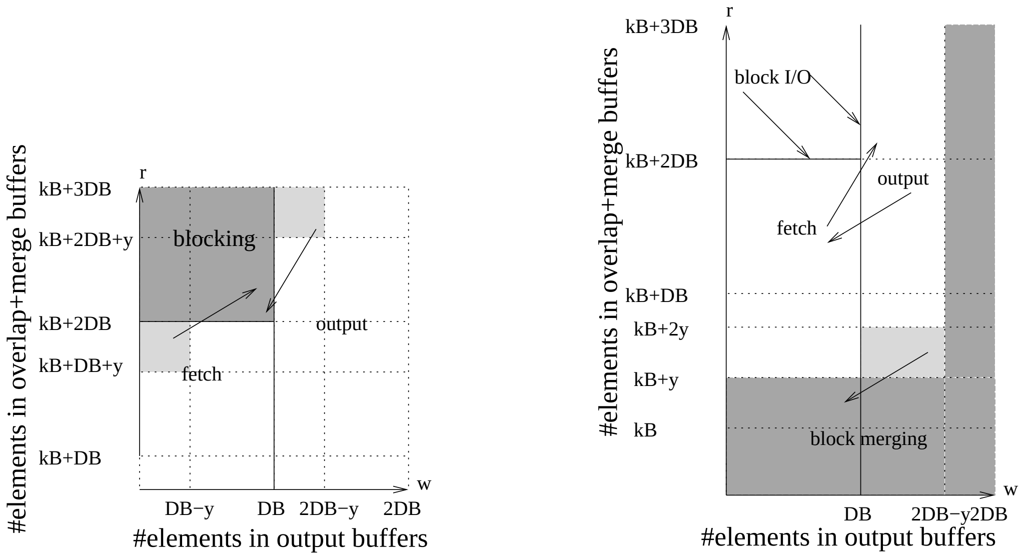
Consider the I/O bound case. We can show that the I/O thread never blocks: Consider figure 7.5. The region in which the I/O thread would block is colored dark grey. Here, there are too few elements in the output buffers (writing them to disk would be inefficient) and the prefetch and merge buffers are still full. The blocking region can only be reached from the light grey areas: Merge y elements into the output buffers and either refill overlap buffers with DB elements[2] elements (left arrow) or write DB items from the output buffers to disk (right arrow). But these two areas themselves cannot be reached at all by the two possible operations; therefore, the I/O thread never blocks.

Analogously, the merging thread does not block in the compute bound case. Here we have two zones of possible blocking: More than $2DB - y$ elements in the output buffers or less than $kB + y$ elements in the merge and prefetch buffers [3]. Again, the constraints on possible transitions in this diagram lead to a single region from where the blocking areas can be reached which is itself unreachable.

[1] Note that $y \leq DB$ in all interesting cases as otherwise we have an I/O-bound algorithm but merging is so fast that there is always enough room to fetch D blocks.

[2] At the same time y are removed by merging resulting in a total increase of $DB - y$.

[3] y elements are not sufficient in general because these may not be the smallest; but if there are more than kB elements in merge and prefetch buffers, at least one element can be written to the output buffer.



I/O bound case compute bound case

Figure 7.5: The limiting thread never blocks

Overlapping is not always possible. Consider the k runs in figure 7.6. While merging them, there is no work to do for the prefetching thread. Then, the current blocks of each run will out of elements simultaneously and merging can only continue after both buffers are refilled.

$$\begin{array}{c}1\ \boxed{1^{B-1}2}\boxed{3^{B-1}4}\boxed{5^{B-1}6}\cdots\\ \vdots \\ k\ \boxed{1^{B-1}2}\boxed{3^{B-1}4}\boxed{5^{B-1}6}\cdots\end{array}$$

Figure 7.6: Overlapping for these k runs is not possible

As internal work influences running time, we need a fast solution for the most compute intense step during merging: A *Tournament Tree* (or Loser Tree) is a specialized data structure for finding the smallest element of all runs, see Figure 7.7. For $k = 2^K$, it is a complete binary tree with K levels, where each leaf contains the currently smallest element of one run. Each internal node contains the 'loser' (i.e., the greater) of the 'competition' between its two child nodes. Above the root node, we store the global winner along with a pointer to the corresponding run. After writing this element to the output buffer, we simply have to move the next element of its run up until there is a new global winner. Compared to general purpose data structures like binary heaps, we get exactly $\log k$ comparisons (no hidden constant factors). Similar to the implicit search trees we used for Sample Sort, Tournament Trees can be implemented as arrays where finding the parent node simply maps to an index shift to the right. The inner loop for moving from leaf to root can be unrolled and contains predictable load instructions and index computations allowing exploitation of instruction parallelism, see Algorithm 7.8.

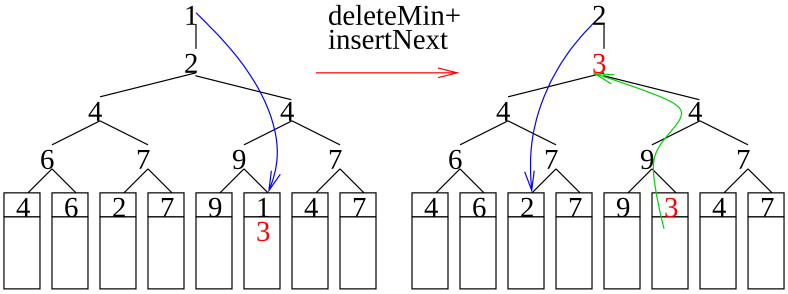


Figure 7.7: A tournament tree

```
for (int i=(winnerIndex+kReg)>>1; i>0; i>>=1){
  currentPos = entry + i;
  currentKey = currentPos->key;
  if (currentKey < winnerKey) {
    currentIndex      = currentPos->index;
    currentPos->key   = winnerKey;
    currentPos->index = winnerIndex;
    winnerKey         = currentKey;
    winnerIndex       = currentIndex;}}
```

Figure 7.8: Inner loop of Tournament Tree computation

7.6 Experiments

Experiments on Multiway Merge Sort were performed in 2001 on a 2 × 2GHz Xeon × 2 threads machine (Intel IV with Netburst) with several 66 MHz PCI-buses, 4 fast IDE controllers (Promise Ultra100 TX2) and 8 fast IDE disks (IBM IC35L080AVVA07). This inexpensive (mid 2002) setup gave a high I/O-bandwidth of 360 MB/s. The keys consisted of 16 GByte random 32 bit integers, run size was 256 MByte, block size B was 2MB (if not otherwise mentioned).

Figure 7.9 shows the running time for different element sizes (for a constant total data volume of 16 GByte). The smaller the elements, the costlier becomes internal work, especially during run formation (there are more elements to sort). With a high I/O throughput and intelligent prefetching algorithms, I/O wait time

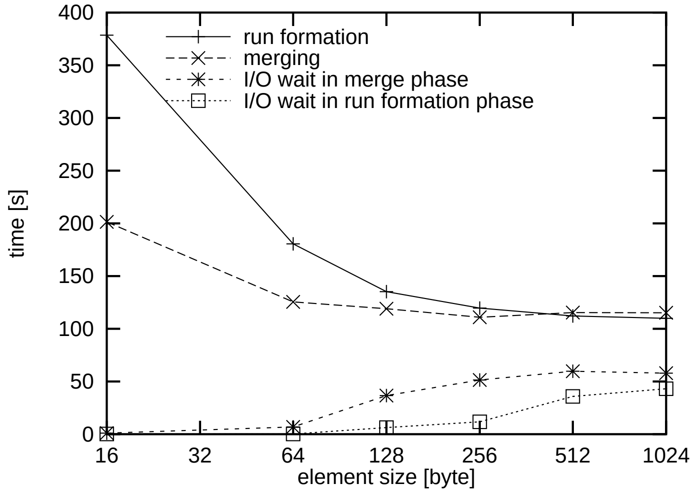


Figure 7.9: Multiway Merge Sort with different element sizes

never makes up for more than half of the total running time. This proves the point that overlapping and tuning of internal work are important.

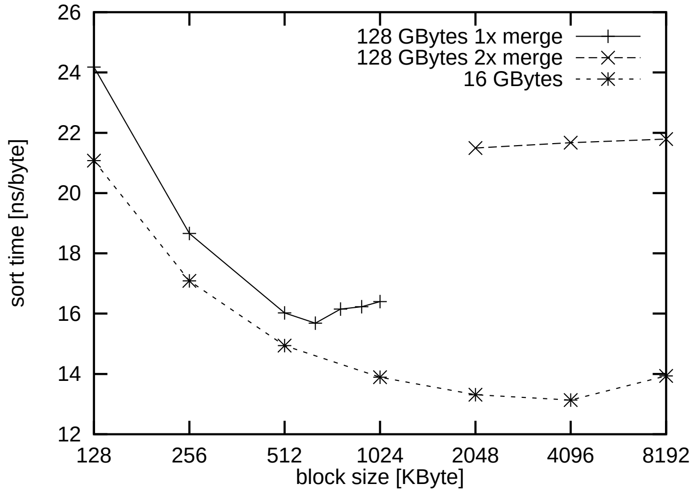


Figure 7.10: Performance using different block sizes

What is a good block size B? An intuitive approach would link B to the size of a physical disk block. However, figure 7.10 shows that B is no technology constant but a tuning parameter: A larger B is better (as it reduces the amortized costs of $\mathrm{O}(1/B)$ I/Os per element), as long as the resulting smaller k still allows for a single merge phase (see curve for 128GB).

7.7 External Priority Queues

The material on external priority queues was first published in [76].

We now study a variant of external priority queues which are called *sequence heaps*[4]: Merging k sorted sequences into one sorted sequence (k-way merging) is an I/O efficient subroutine used for sorting – we saw this in chapter 7.4. The basic idea of sequence heaps is to adapt k-way merging to the related but more dynamical problem of priority queues.

Let us start with the simple case, that at most km insertions take place where m is the size of a buffer that fits into fast memory. Then the data structure could consist of k sorted sequences of length up to m. We can use k-way merging for deleting a batch of the m smallest elements from k sorted sequences.

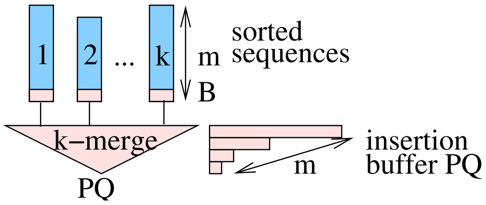


Figure 7.11: A simple external PQ for $n < km$

A separate binary heap with capacity m allows an arbitrary mix of insertions and deletions by holding the recently inserted elements. Deletions have to check whether the smallest element has to come from this *insertion buffer*. When this buffer is full, it is sorted, and the resulting sequence becomes one of the sequences for the k-way merge.

How can we generalize this approach to handle more than km elements? We cannot increase m beyond M, since the insertion heap would not fit into fast memory. We cannot arbitrarily increase k, since eventually k-way merging would start to incur cache faults. Sequence heaps make room by merging all the k sequences producing a larger sequence of size up to km.

Now the question arises how to handle the larger sequences. Sequence heaps employ R *merge groups* $G_1, \ldots, G_R$ where G_i holds up to k sequences of size up to mk^{i-1}. When group G_i overflows, all its sequences are merged, and the resulting sequence is put into group G_{i+1}.

Each group is equipped with a *group buffer* of size m to allow batched deletion from the sequences. The smallest elements of these buffers are deleted in batches of size $m' \ll m$. They are stored in the *deletion buffer*. Fig. 7.12 summarizes the data structure.

[4] By replacing "I/O" with "cache fault", we can also use this approach one level higher in the memory hierarchy.

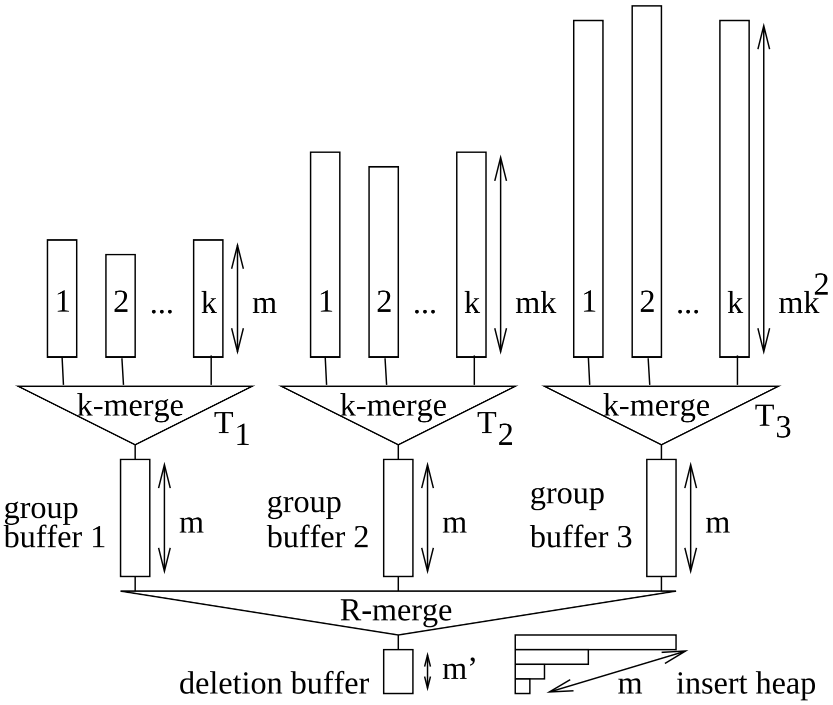


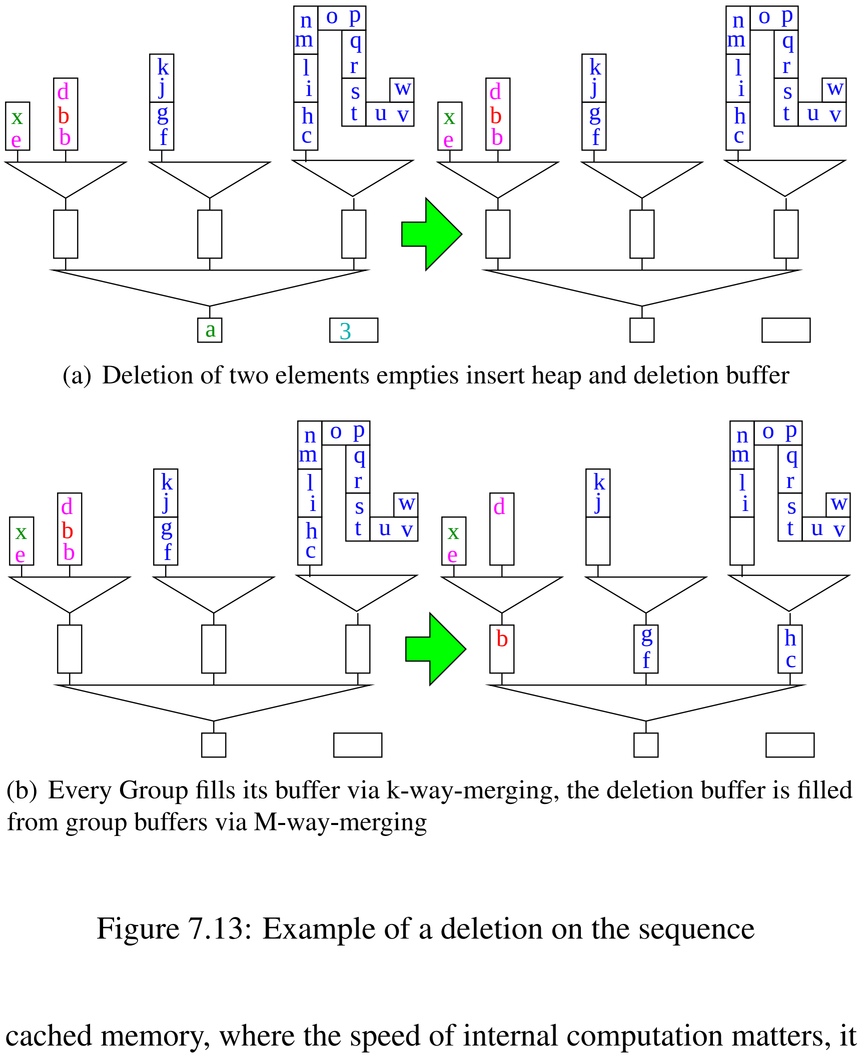
Figure 7.12: Overview of the complete data structure for $R = 3$ merge groups

We now have enough information to understand how deletion and insertion operations are going to work:

DeleteMin: The smallest elements of the deletion buffer and insertion buffer are compared, and the smaller one is deleted and returned. If this empties the deletion buffer, it is refilled from the group buffers using an R-way merge. Before the refill, group buffers with less than m' elements are refilled from the sequences in their group (if the group is nonempty). Figure 7.13 gives an example of a deletion operation.

DeleteMin works correctly provided the data structure fulfills the heap property, i.e., elements in the group buffers are not smaller than elements in the deletion buffer, and in turn, elements in a sorted sequence are not smaller than the elements in the respective group buffer. Maintaining this invariant is the main difficulty for implementing insertion.

Insert: New elements are inserted into the insert heap. When its size reaches m, its elements are sorted (e.g. using merge sort or heap sort). The result is then merged with the concatenation of the deletion buffer and the group buffer 1. The smallest resulting elements replace the deletion buffer and group buffer 1. The remaining elements form a new sequence of length at most m. The new sequence is finally inserted into a free slot of group G_1. If there is no free slot initially, G_1 is emptied by merging all its sequences into a single sequence of size at most km, which is then put into G_2. The same strategy is used recursively to free higher level groups when necessary. When group G_R overflows, R is incremented and a new group is created. When a sequence is moved from one group to the other, the heap property may be violated. Therefore, when G_1 through G_i have been emptied, the group buffers 1 through $i + 1$ are merged, and put into G_1. An example insertion operation is shown in Figures 7.14 and 7.15.

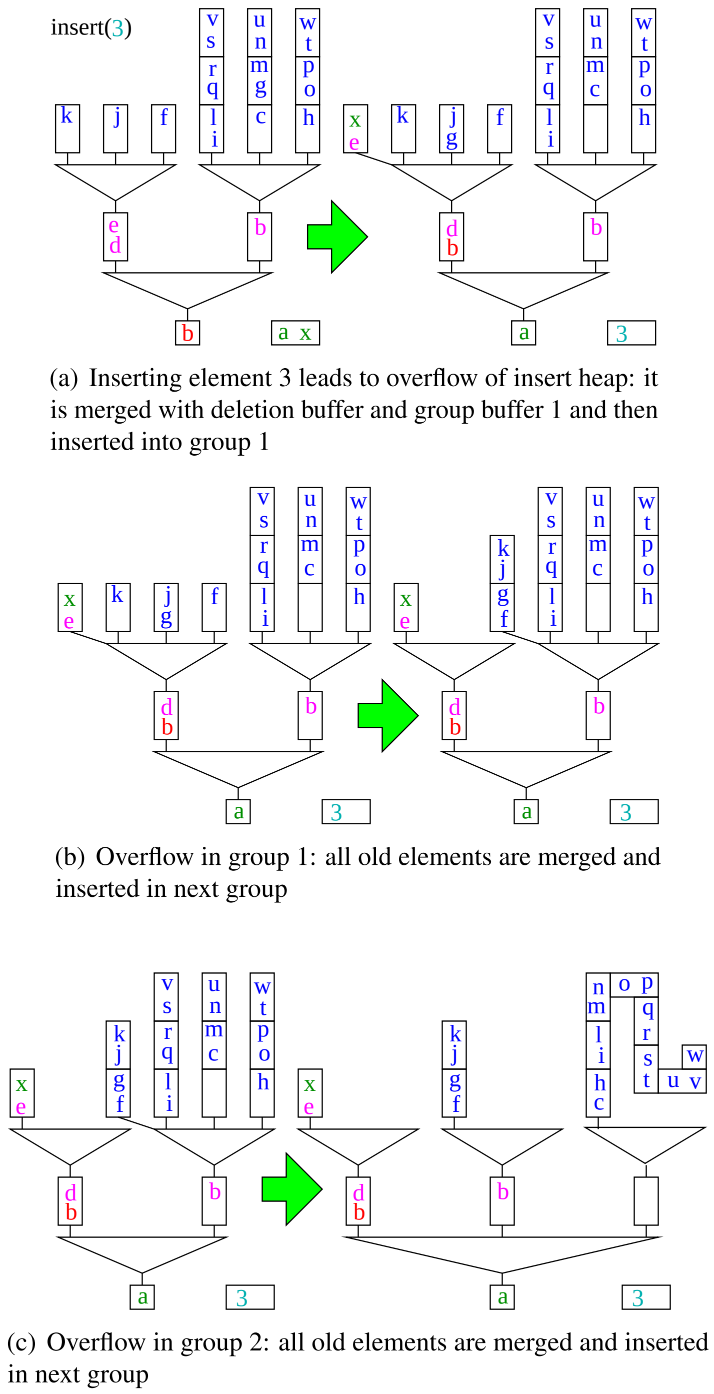


(a) Deletion of two elements empties insert heap and deletion buffer

(b) Every Group fills its buffer via k-way-merging, the deletion buffer is filled from group buffers via M-way-merging

Figure 7.13: Example of a deletion on the sequence

For cached memory, where the speed of internal computation matters, it is also crucial how to implement the operation of k-way merging. How this can be done in an efficient way is described in the chapter about Sorting (7.5).

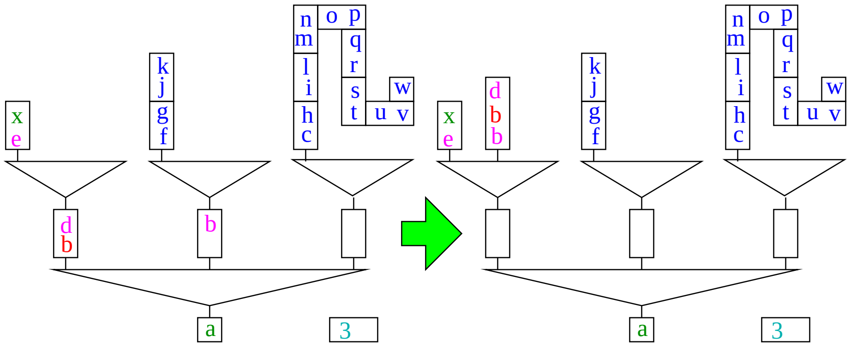


(a) Inserting element 3 leads to overflow of insert heap: it is merged with deletion buffer and group buffer 1 and then inserted into group 1

(b) Overflow in group 1: all old elements are merged and inserted in next group

(c) Overflow in group 2: all old elements are merged and inserted in next group

Figure 7.14: Example of an insertion on the sequence heap (part 1)



(d) Group buffers are invalid now: merge and inserted them to group 1

Figure 7.15: Example of an insertion on the sequence heap (part 2)

Analysis

We will now give a sketch for the I/O analysis of our priority queues. Let i denote the number of insertions and an upper bound to the number of deleteMin operations.

First note that Group G_i can overflow at most every $m(k^i - 1)$ insertions: The only complication is the slot in group G_1 used for invalid group buffers. Nevertheless, when groups G_1 through G_i contain k sequences each, this can only happen if

$$\sum_{j=1}^{R} m(k-1)k^{j-1} = m(k^i - 1)$$

insertions have taken place. Therefore, $R = \lceil \log_k \frac{I}{m} \rceil$ groups suffice.

Now consider the I/Os performed for an element moving on the following *canonical* data path: It is first inserted into the insert buffer and then written to a sequence in group G_1 in a batched manner, i.e., $1/B$ I/Os are charged to the insertion of this element. Then it is involved in emptying groups until it arrives in group G_R. For each emptying operation, the element is involved into one batched read and one batched write, i.e., it is charged with $2(R-1)/B$ I/Os for tree emptying operations. Eventually, the element is read into group buffer R yielding a charge of $1/B$ I/Os for. All in all, we get a charge of $2R/B$ I/Os for each insertion.

What remains to be shown (and is ommited here) is that the remaining I/Os only contribute lower order terms or replace I/Os done on the canonical path. For example, we save I/Os when an element is extracted before it reaches the last group. We use the costs charged for this to pay for swapping the group buffers in and out. Eventually, we have $\mathrm{O}(sort(I))$ I/Os.

In a similar fashion, we can show that I operations inflict $I \log I$ key comparisons on average. As for sorting, this is a good measure for the internal work, since in efficient implementations of priority queues for the comparison model, this number is close to the number of unpredictable branch instructions (whereas loop control branches are usually well predictable by the hardware or the compiler), and the number of key comparisons is also proportional to the number of memory accesses. These two types of operations often have the largest impact on the execution time, since they are the most severe limit to instruction parallelism in a super-scalar processor.

Experiments

We now present the results of some experiments conducted to compare our *sequence heap* with other priority queue implementations. Random 32 bit integers were used as keys for another 32 bits of associated information. The operation sequence used was $(\texttt{Insert} - \texttt{deleteMin} - \texttt{Insert})^N(\texttt{deleteMin} - \texttt{Insert} - \texttt{deleteMin})^N$. The choice of this sequence is nontrivial as it can have measurable influence (factor two and more) on the performance. Figure 7.17 show this: Here we have the sequence $(\texttt{Insert}\ (\texttt{deleteMin}\ \texttt{Insert})^s)^N\ (\texttt{deleteMin}\ (\texttt{Insert}\ \texttt{deleteMin})^s)^N$ for several values of s. For larger s, the performance gets better when N is large enough. This can be explained with a “locality effect“: New elements tend to be smaller than most old elements (the smallest of the old elements have long been removed before). Therefore, many elements never make it into group G_1 let alone the groups for larger sequences. Since most work is performed while emptying groups, this work is saved. So that these instances should come close to the worst case. To make clear that sequence heaps are nevertheless still much better than binary or 4-ary heaps, Figure 7.17 additionally contains their timing for $s = 0$.

The parameters chosen for the experiments where $m' = 32$, $m = 256$ and $k = 128$ on all machines tried. While there were better settings for individual machines, these global values gave near optimal performance in all cases.

7.8 Semiexternal Kruskal’s Algorithm

A first step for an algorithm that can cope with huge graphs stored on disk is a semiexternal algorithm: We use Kruskal’s Algorithm but incorporate an external sorting algorithm. We then just have to scan the edges and maintain the union-find array in main memory. This only requires one 32 bit word per node to store up to $0..2^{32} - 32 = 4\,294\,967\,264$ nodes. There exist asymptotically better algorithms but these come with discouraging constant factors and significantly larger data structures.

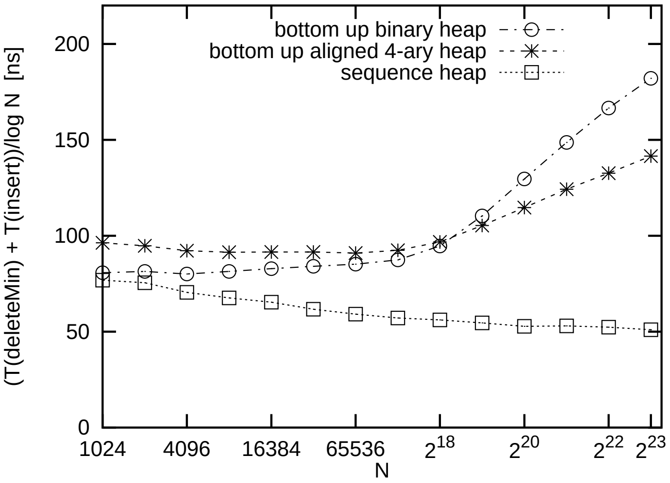


Figure 7.16: Runtime comparison for several PQ implementations (on a 180MHz MIPS R10000)

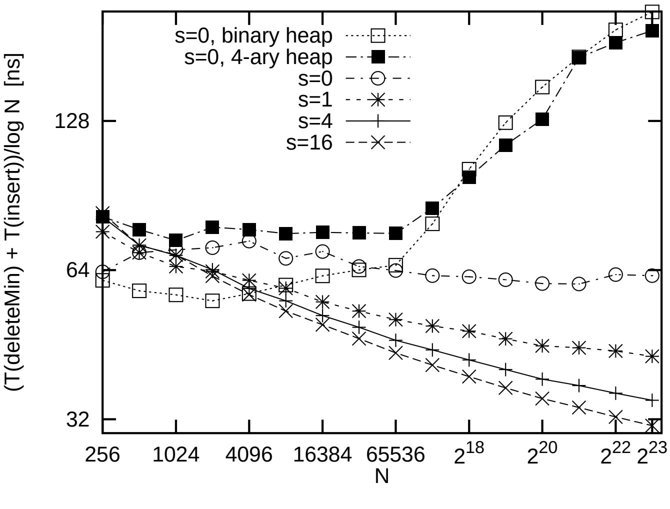


Figure 7.17: Runtime comparison for different operation sequences

Chapter 8

Parallel Algorithms

This chapter is mainly based on [49].

8.1 Introduction

Numerous programming languages and libraries have been developed for explicit parallel programming. These differ in their view of the address space that they make available to the programmer, the degree of synchronization imposed on concurrent activities, and the multiplicity of programs. The message-passing programming paradigm is one of the oldest and most widely used approaches for programming parallel computers. Its roots can be traced back in the early days of parallel processing, and its wide-spread adoption can be attributed to the fact that it imposes minimal requirements on the underlying hardware.

8.2 Message-Passing Model/Paradigm

There are two key attributes that characterize the message-passing programming paradigm. The first is that it assumes a partitioned address space, and the second is that it supports only explicit parallelization.

The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space. Instances of such a view come naturally from clustered workstations and non-shared address space multicomputers. There are two immediate implications of a partitioned address space. First, each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed. This adds complexity to programming, but encourages locality of access, which is critical for achieving high performance on non-UMA architecture, since a processor can access its local data much faster than non-local data on such architectures. The second implication is that all interactions (read-only or read/write) require cooperation of two processes, the process that has the data and the process that wants to access the data. This requirement for cooperation adds a great deal of complexity for a number of reasons. The process that has the data must participate in the interaction even if it has no logical connection to the events at the requesting process. In certain circumstances, this requirement leads to unnatural programs. In particular, for dynamic and/or unstructured interactions, the complexity of the code written for this type of paradigm can be very high for this reason. However, a primary advantage of explicit two-way interactions is that the programmer is fully aware of all the costs of non-local interactions, and is more likely to think about algorithms (and mappings) that minimize interactions. Another major advantage of this type of programming paradigm is that it can be efficiently implemented on a wide variety of architectures.

The message-passing programming paradigm requires that the parallelism is coded explicitly by the programmer. That is, the programmer is responsible for analyzing the underlying serial algorithm/application and identifying ways by which he or she can decompose the computations and extract concurrency. As a result, programming using the message-passing paradigm tends to be hard and intellectually demanding. However, on the other hand, properly written message-passing programs can often achieve very high performance and scale to a very large number of processes.

Message-passing programs are often written using the asynchronous or loosely synchronous paradigms. In the asynchronous paradigm, all concurrent tasks execute asynchronously. This makes it possible to implement any parallel algorithm. However, such programs can be harder to reason about, and can have nondeterministic behavior due to race conditions. Loosely synchronous programs are a good compromise between these two extremes. In such programs, tasks or subsets of tasks synchronize to perform interactions. However, between these interactions, tasks execute completely asynchronously. Since the interaction happens synchronously, it is still quite easy to reason about the program. Many of the known parallel algorithms can be naturally implemented using loosely synchronous programs.

In its most general form, the message-passing paradigm supports execution of a different program on each of the p processes. This provides the ultimate flexibility in parallel programming, but makes the job of writing parallel programs effectively unscalable. For this reason, most message-passing programs are written using the single program multiple data (SPMD) approach. In SPMD programs the code executed by different processes is identical except for a small number of processes (e.g., the “root” process). This does not mean that the processes work in lock-step. In an extreme case, even in an SPMD program, each process could execute a different code (the program contains a large case statement with code for each process). But except for this degenerate case, most processes execute the same code. SPMD programs can be loosely synchronous or completely asynchronous.

8.3 Communication Cost in Parallel Machines

The time taken to communicate a message between two nodes in a network is the sum of the time to prepare a message for transmission and the time taken by the message to traverse the network to its destination. The principal parameters that determine the communication latency are as follows:

- Startup time T_{start}: The startup time is the time required to handle a message at the sending and receiving nodes. This includes the time to prepare the message (adding header, trailer, and error correction information), the time to execute the routing algorithm, and the time to establish an interface between the local node and the router. This delay is incurred only once for a single message transfer.

- Per-hop time T_{hop}: After a message leaves a node, it takes a finite amount of time to reach the next node in its path. The time taken by the header of a message to travel between two directly-connected nodes in the network is called the per-hop time. It is also known as node latency. The per-hop time is directly related to the latency within the routing switch for determining which output buffer or channel the message should be forwarded to.

- Per-byte transfer time T_{byte}: If the channel bandwidth is r bytes per second, then each word takes time $T_{\text{byte}} = 1/r$ to traverse the link. This time is called the per-byte transfer time. This time includes network as well as buffering overheads.

This implies that in order to optimize the cost of message transfers, we would need to:

- Communicate in bulk. That is, instead of sending small messages and paying a startup costs for each one, we want to aggregate small messages into a single large message and amortize the startup latency across a larger message. This is because on typical platforms such as clusters and message-passing machines, the value of T_{start} is much larger than those of T_{hop} or T_{byte}.

- Minimize the volume of data. To minimize the overhead paid in terms of per-byte transfer time T_{byte}, it is desirable to reduce the volume of data communicated as much as possible.

- Minimize distance of data transfer. Minimize the number of hops that a message must traverse.

While the first two objectives are relatively easy to achieve, the task of minimizing distance of communicating nodes is difficult, and in many cases an unnecessary burden on the algorithm designer. This is a direct consequence of the following characteristics of parallel platforms and paradigms:

- In many message-passing libraries such as MPI, the programmer has little control on the mapping of processes onto physical processors. In such paradigms, while tasks might have well defined topologies and may communicate only among neighbors in the task topology, the mapping of processes to nodes might destroy this structure.

- Many architectures rely on randomized (two-step) routing, in which a message is first sent to a random node from source and from this intermediate node to the destination. This alleviates hot-spots and contention on the network. Minimizing number of hops in a randomized routing network yields no benefits.

- The per-hop time T_{hop} is typically dominated either by the startup latency T_{start} for small messages or by per-byte component ℓT_{byte} for large messages ℓ. Since the maximum number of hops in most networks is relatively small, the per-hop time can be ignored with little loss in accuracy.

All of these point to a simpler cost model in which the cost of transferring a message between two nodes on a network is given by:

$$T_{\text{comm}}(\ell) = T_{\text{start}} + \ell T_{\text{byte}}$$

This expression has significant implications for architecture-independent algorithm design as well as for the accuracy of runtime predictions. Since this cost model implies that it takes the same amount of time to communicate between any pair of nodes, it corresponds to a completely connected network. Instead of designing algorithms for each specific architecture (for example, a mesh, hypercube, or tree), we can design algorithms with this cost model in mind and port it to any target parallel computer.

This raises the important issue of loss of accuracy (or fidelity) of prediction when the algorithm is ported from our simplified model (which assumes a completely connected network) to an actual machine architecture. If our initial assumption that the T_{hop} term is typically dominated by the T_{start} or T_{byte} terms is valid, then the loss in accuracy should be minimal.

However, it is important to note that our basic cost model is valid only for uncongested networks. Architectures have varying thresholds for when they get congested; i.e., a linear array has a much lower threshold for congestion than a hypercube. Furthermore, different communication patterns congest a given network to different extents. Consequently, our simplified cost model is valid only as long as the underlying communication pattern does not congest the network.

8.4 Performance metrics

It is important to study the performance of parallel programs with a view to determining the best algorithm, evaluating hardware platforms, and examining the benefits from parallelism. A number of metrics have been used based on the desired outcome of performance analysis.

8.4.1 Execution Time

The serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer. The parallel runtime is the time that elapses from the moment a parallel computation starts to the moment the last processing element finishes execution. We denote the serial runtime by T_{seq} and the parallel runtime by $T(p)$.

8.4.2 Speedup

When evaluating a parallel system, we are often interested in knowing how much performance gain is achieved by parallelizing a given application over a sequential implementation. Speedup is a measure that captures the relative benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with p identical processing elements. We denote speedup by the symbol S.

For a given problem, more than one sequential algorithm may be available, but all of these may not be equally suitable for parallelization. When a serial computer is used, it is natural to use the sequential algorithm that solves the problem in the least amount of time. Given a parallel algorithm, it is fair to judge its performance with respect to the fastest sequential algorithm for solving the same problem on a single processing element. Sometimes, the asymptotically fastest sequential algorithm to solve a problem is not known, or its runtime has a large constant that makes it impractical to implement. In such cases, we take the fastest known algorithm that would be a practical choice for a serial computer to be the best sequential algorithm. We compare the performance of a parallel algorithm to solve a problem with that of the best sequential algorithm to solve the same problem. We formally define the speedup S as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processing elements. The p processing elements used by the parallel algorithm are assumed to be identical to the one used by the sequential algorithm.

8.4.3 Superlinear speedup

Theoretically, speedup can never exceed the number of processing elements, p. If the best sequential algorithm takes T_{seq} units of time to solve a given problem on a single processing element, then a speedup of p can be obtained on p processing elements if none of the processing elements spends more than time T_{seq}/p. A speedup greater than p is possible only if each processing element spends less than time T_{seq}/p solving the problem. In this case, a single processing element could emulate the p processing elements and solve the problem in fewer than T_{seq} units of time. This is a contradiction because speedup, by definition, is computed with respect to the best sequential algorithm. If T_{seq} is the serial runtime of the algorithm, then the problem cannot be solved in less than time T_{seq} on a single processing element.

In practice, a speedup greater than p is sometimes observed (a phenomenon known as superlinear speedup). This usually happens when the work performed by a serial algorithm is greater than its parallel formulation or due to hardware features that put the serial implementation at a disadvantage. For example, the data for a problem might be too large to fit into the cache of a single processing element, thereby degrading its performance due to the use of slower memory elements. But when partitioned among several processing elements, the individual data-partitions would be small enough to fit into their respective processing elements' caches.

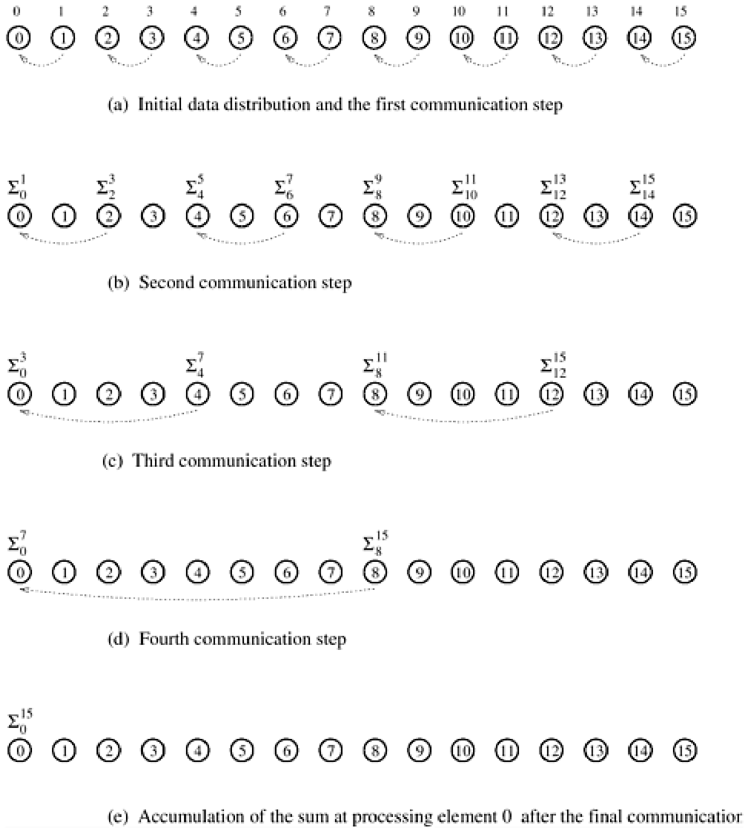
8.4.4 Efficiency

Only an ideal parallel system containing p processing elements can deliver a speedup equal to p. In practice, ideal behavior is not achieved because while executing a parallel algorithm, the processing elements cannot devote 100% of their time to the computations of the algorithm. Efficiency is a measure of the fraction of time for which a processing element is usefully employed; it is defined as the ratio of speedup to the number of processing elements. In an ideal parallel system, speedup is equal to p and efficiency is equal to one. In practice, speedup is less than p and efficiency is between zero and one, depending on the effectiveness with which the processing elements are utilized. We denote efficiency by the symbol E. Mathematically, it is given by

$$E = \frac{S}{p}$$

8.4.5 Example: Adding n numbers using n processing elements

Consider the problem of adding n numbers by using $p = n$ processing elements. Initially, each processing element is assigned one of the numbers to be added and, at the end of the computation, one of the processing elements stores the sum of all the numbers. Assuming that n is a power of two, we can perform this operation in $\log n$ steps by propagating partial sums up a logical binary tree of processing elements. Figure 8.1 illustrates the procedure for $n = 16$. The processing elements are labeled from 0 to 15. Similarly, the 16 numbers to be added are labeled from 0 to 15. The sum of numbers with consecutive labels from i to j is denoted by $\sum_i^j$.


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Σ_0^1 Σ_2^3 Σ_4^5 Σ_6^7 Σ_8^9 Σ_{10}^{11} Σ_{12}^{13} Σ_{14}^{15}
Σ_0^3 Σ_4^7 Σ_8^{11} Σ_{12}^{15}
Σ_0^7 Σ_8^{15}
Σ_0^{15}

(a) Initial data distribution and the first communication step

(b) Second communication step

(c) Third communication step

(d) Fourth communication step

(e) Accumulation of the sum at processing element 0 after the final communicatior

Figure 8.1: Computing the global sum of 16 partial sums using 16 processing elements.$\sum_i^j$ denotes the sum of numbers with consecutive labels from i to j.

Each step shown in Figure 8.1 consists of one addition and the communication of a single word. The addition can be performed in some constant time, say $T_{c'}$, and the communication of a single word can be performed in time $T_{\text{start}} + T_{\text{byte}}$. Therefore, the addition and communication operations take a constant amount of time. Thus,

$$T(p) = \Theta(\log n)$$

Since the problem can be solved in $\Theta(n)$ time on a single processing element, its speedup is

$$S = \Theta(\frac{n}{\log n})$$

From this equation and the preceding definition, the efficiency of the algorithm for adding n numbers on n processing elements is

$$E = \frac{\Theta(\frac{n}{\log n})}{n} = \Theta(\frac{1}{\log n})$$

8.4.6 Example: Adding n numbers using p processing elements

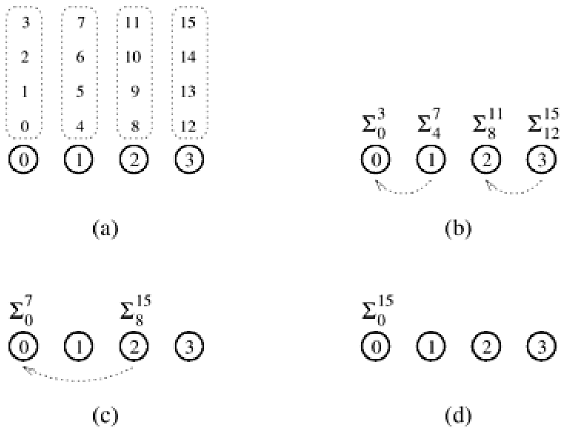


Figure 8.2: Computing the sum of 16 numbers using four processing elements.

An alternate method for adding n numbers using p processing elements is illustrated in Figure 8.2 for $n = 16$ and $p = 4$. In the first step of this algorithm, each processing element locally adds its n/p numbers in time $T_{\text{seq}} = \Theta(n/p)$. Now the problem is reduced to adding the p partial sums on p processing elements, which can be done in time $\Theta(\log p)$ by the method described in the previous example. The parallel runtime of this algorithm is

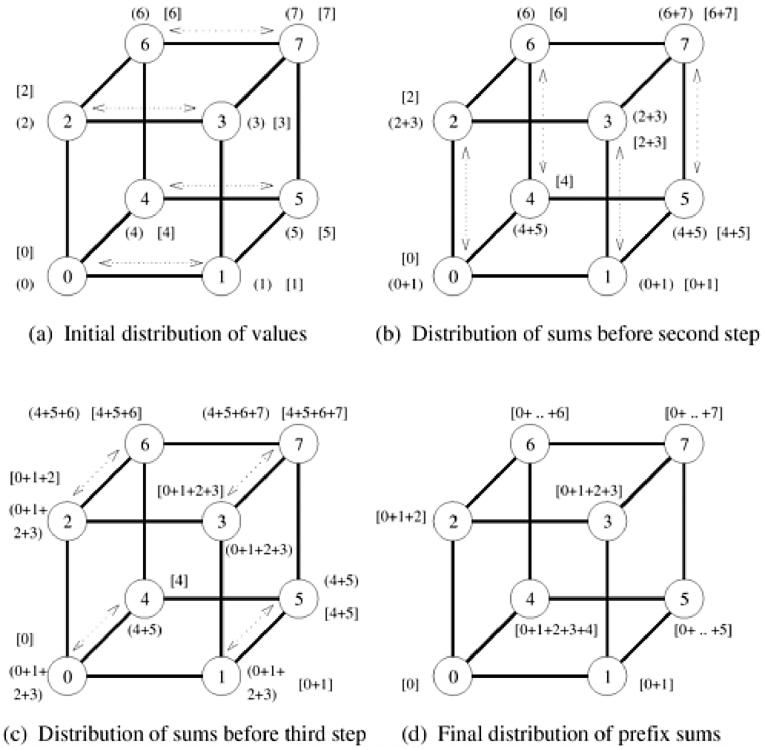
$$T(p) = \Theta(n/p + \log p)$$

the efficiency is

$$E = \frac{T_{\text{seq}}(n)}{p(T_{\text{seq}}(n/p) + \Theta(\log p))} = \frac{1}{1 + \Theta(p\log(p))\,/n} = 1 - \Theta\left(\frac{p\log p}{n}\right)$$

8.5 Prefix Sums on a Hypercube Network

In the following, we describe prefix sum operations and derive expressions for their time complexity. We assume that the communication time between any pair of nodes is practically independent of of the number of intermediate nodes along the paths between them. We also assume that the communication links are bidirectional; that is, two directly-connected nodes can send messages of size m to each other simultaneously in time $T_{\text{start}} + \ell T_{\text{byte}}$. We assume a single-port communication model, in which a node can send a message on only one of its links at a time. Similarly, it can receive a message on only one link at a time. However, a node can receive a message while sending another message at the same time on the same or a different link.



(a) Initial distribution of values

(b) Distribution of sums before second step

(c) Distribution of sums before third step

(d) Final distribution of prefix sums

Figure 8.3: Computing the sum of 16 numbers using four processing elements.

Algorithm 1 PREFIX_SUMS_HCUBE(my_id, my_number, d, $result$)

```
$result := my\_number$
$msg := result$
for $i := 0$ to $d - 1$ do
    $partner := my\_id$ XOR $2^i$
    send $msg$ to partner
    receive $number$ from partner
    $msg := msg + number$
    if $partner < my\_id$ then
        $result := result + number$
    end if
end for
```

Algorithm 1 gives a procedure for implementing prefix sums on a d-dimensional hypercube. Communication starts from the lowest dimension of the hypercube and then proceeds along successively higher dimensions (Line 5). In each iteration, nodes communicate in pairs so that the labels of the nodes communicating with each other in the i th iteration differ in the ith least significant bit of their binary representations (Line 6).The node with label k uses information from only the k-node subset of those nodes whose labels are less than or equal to k. To accumulate the correct prefix sum, every node maintains an additional result buffer. This buffer is denoted by square brackets in Figure 8.3. After an iteration's communication steps, each node adds the data it receives during that iteration to its resident data (denoted by parentheses in the figure) (Line 9). This sum is transmitted in the following iteration. The content of an incoming message is added to the result buffer only if the message comes from a node with a smaller label than that of the recipient node (Line 10).

On a p-node hypercube, the size of each message exchanged in the i th of the $\log p$ steps is ℓ. It takes a pair of nodes time $T_{\text{start}} + \ell T_{\text{byte}}$ to send and receive messages from each other during the i th step. Hence, the time to complete the entire procedure is

$$T = \log p(T_{\text{start}} + \ell T_{\text{byte}})$$

8.6 Quicksort

This section examines the quicksort algorithm, which has an average complexity of $\Theta(n \log n)$. Quicksort is one of the most common sorting algorithms for sequential computers because of its simplicity, low overhead, and optimal average complexity.

8.6.1 Sequential Algorithm

Quicksort is a divide-and-conquer algorithm that sorts a sequence by recursively dividing it into smaller subsequences. Assume that the n-element sequence to be sorted is stored in the array $A[1 \dots n]$. Quicksort consists of two steps: divide and conquer. During the divide step, a sequence $A[q \dots r]$ is partitioned (rearranged) into two nonempty subsequences $A[q \dots s]$ and $A[s{+}1 \dots r]$ such that each element of the first subsequence is smaller than or equal to each element of the second subsequence. During the conquer step, the subsequences are sorted by recursively applying quicksort. Since the subsequences $A[q \dots s]$ and $A[s + 1 \dots r]$ are sorted and the first subsequence has smaller elements than the second, the entire sequence is sorted.

How is the sequence $A[q \dots r]$ partitioned into two parts – one with all elements smaller than the other? This is usually accomplished by selecting one element x from $A[q \dots r]$ and using this element to partition the sequence $A[q \dots r]$ into two parts – one with elements less than or equal to x and the other with elements greater than x. Element x is called the pivot. The quicksort algorithm is presented in Algorithm 2. This algorithm arbitrarily chooses the first element of the sequence $A[q \dots r]$ as the pivot. The operation of quicksort is illustrated in Figure 8.4.

Algorithm 2 QUICKSORT(A, q, r)

```
if $q < r$ then
    $x := A[q]$
    $s := q$
    for $i := q + 1$ to $r$ do
        if $A[i] \leq x$ then
            $s := s + 1$
            swap($A[s]$, $A[i]$)
        end if
    end for
    swap($A[q], A[s]$)
    QUICKSORT($A, q, s$)
    QUICKSORT($A, s + 1, r$)
end if
```

The complexity of partitioning a sequence of size k is $\Theta(k)$. Quicksort's performance is greatly affected by the way it partitions a sequence. Consider the case in which a sequence of size k is split poorly, into two subsequences of sizes 1 and $k - 1$. The run time in this case is given by the recurrence relation $T(n) = T(n - 1) + \Theta(n)$, whose solution is $T(n) = \Theta(n^2)$. Alternatively, consider the case in which the sequence is split well, into two roughly equal-size

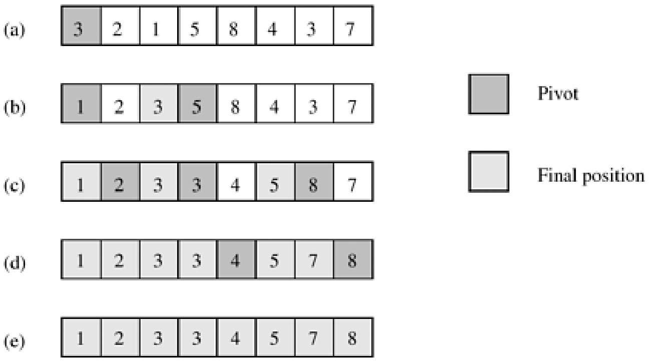


Figure 8.4: Computing the sum of 16 numbers using four processing elements.

.

subsequences of and elements. In this case, the run time is given by the recurrence relation $T(n) = 2T(n/2) + \Theta(n)$, whose solution is $T(n) = \Theta(n \log n)$. The second split yields an optimal algorithm. Although quicksort can have $O(n^2)$ worst-case complexity, its average complexity is significantly better; the average number of compare-exchange operations needed by quicksort for sorting a randomly-ordered input sequence is $1.44n \log n$, which is asymptotically optimal. There are several ways to select pivots. For example, the pivot can be the median of a small number of elements of the sequence, or it can be an element selected at random. Some pivot selection strategies have advantages over others for certain input sequences.

8.6.2 Parallelizing Quicksort

Quicksort can be parallelized in a variety of ways. First, consider a naive parallel formulation in the context of recursive decomposition. During each call of QUICKSORT (see Algorithm 2), the array is partitioned into two parts and each part is solved recursively. Sorting the smaller arrays represents are two completely independent subproblems that can be solved in parallel. Therefore, one way to parallelize quicksort is to execute it initially on a single process; then, when the algorithm performs its recursive calls, assign one of the subproblems to another process. Now each of these processes sorts its array by using quicksort and assigns one of its subproblems to other processes. The algorithm terminates when the arrays cannot be further partitioned. Upon termination, each process holds an element of the array, and the sorted order can be recovered by traversing the processes as we will describe later. This parallel formulation of quicksort uses n processes to sort n elements. Its major drawback is that partitioning the array $A[q \ldots r]$ into two smaller arrays, $A[q \ldots s]$ and $A[s + 1 \ldots$r], is done by a single process. Since one process must partition the original array $A[1 \ldots n]$, the run time of this formulation is bounded below by $\Omega(n)$.

The main limitation of the previous parallel formulation is that it performs the partitioning step serially. As we will see in subsequent formulations, performing partitioning in parallel is essential in obtaining an efficient parallel quicksort. To see why, consider the recurrence equation $T(n) = 2T(n/2) + \Theta(n)$, which gives the complexity of quicksort for optimal pivot selection. The term $\Theta(n)$ is due to the partitioning of the array. Compare this complexity with the overall complexity of the algorithm, $\Theta(n \log n)$. From these two complexities, we can think of the quicksort algorithm as consisting of $\Theta(\log n)$ steps, each requiring time $\Theta(n)$ that of splitting the array. Therefore, if the partitioning step is performed in time $\Theta(1)$, using $\Theta(n)$ processes, it is possible to obtain an overall parallel run time of $\Theta(\log n)$. Hence, parallelizing the partitioning step has the potential to yield a significantly faster parallel formulation.

In the previous paragraph, we hinted that we could partition an array of size n into two smaller arrays in time $\Theta(1)$ by using $\Theta(n)$ processes. However, this is difficult for most parallel computing models. The only known algorithms are for the abstract PRAM models. Because of communication overhead, the partitioning step takes longer than $\Theta(1)$ on realistic shared-address-space and message-passing parallel computers. In the following section we present a parallel formulation for a message-passing platform.

8.6.3 Parallel Formulation for a Message-passing Platform

Let A be an array of n elements that need to be sorted and $p = n$ be the number of processes. The array A is explicitly distributed among the processes and each process stores one element d of A. The labels of the processes define the global order of the sorted sequence. The algorithm starts by selecting a pivot element v, which is broadcast to all processes. Each process P_i, upon receiving the pivot, compares its element d with the pivot. In the next phase, the algorithm first determines which processes will be responsible for recursively sorting the smaller-than-the-pivot elements (i.e., $\leq v$) and which process will be responsible for recursively sorting the larger-than-the-pivot elements (i.e., $> v$). Once this is done, the processes send their elements d to the corresponding processes. After that, the processes are partitioned into the two groups, and the algorithm proceeds recursively.

The method to determine the process where the process P_i should send its element d to, is based on computing the number of processes $j = |\{P_k \mid k \leq i\}|$ that store elements $\leq v$. Thus, is the target process is either $j - 1$ if $d \leq v$, or $p' + i - j$ if $d > v$, where $p' = |\{d \mid d \leq v\}|$. Obviously, j and p' can be computed using a prefix sum operation and a broadcast.

Algorithm 3 theoQSort(d, i, p)

```
if $p = 1$ then
  return
end if
$r :=$ random element from $0 \ldots p-1$ {same value in entire partition}
$v := d@r$ {broadcast pivot}
$f := d \leq v$ {1 if $d$ is on left side, 0 otherwise}
$j := \sum_{k=0}^{i} f@k$ {prefix sum, count elements on left side}
$p' := j@(p-1)$ {broadcast, result is border index}
if $f$ then
  send $d$ to PE $j-1$
else
  send $d$ to PE $p'+i-j$ {$i-j = \sum_{k=0}^{i} d@k > v$}
end if
receive $d$
if $i < p'$ then
  join left partition;   qsort$(d, i, p')$
else
  join right partition;   qsort$(d, i-p', p-p')$
end if
```

Analysis. The amount of time required to split an array of size n is $2 \times \Theta(\log p)$ for broadcasting the pivot element and p', $\Theta(\log p)$ for performing the prefix sum. Thus, the overall complexity for the split is $\Theta(\log p)$. This process is repeated for each of the two subsequences recursively. Assuming that the pivot splits the input into roughly equal size subsequences, the expected recursion length is $\Theta(\log p)$. Thus, the overall complexity of the parallel algorithm is:

$$T(p) = \Theta(\log^2 p)$$

8.7 Mergesort

Given k sorted sequences $S_1, \ldots, S_k$ and a global rank m, we are asked to find splitting positions $i_1, \ldots, i_k$ such that $i_1 + \ldots + i_k = m$ and $\forall j, j' \in 1 \ldots k : S_j[i_j - 1] \leq S_{j'}[i_{j'}]$. This is similar to selection, although in this chapter, we are not interested in the element of global rank m, but rather the splitting positions. We call finding the splitting positions multiway partitioning. Finding the respective element is trivial after multiway partitioning, just take the smallest element right of the split. Merging two sorted sequences S_1 and S_2 is usually understood as efficiently coalescing them into one sorted sequence. Its generalization to multiple sequences $S_1, \ldots, S_k$ is called multiway merging.

It is often used instead of repeated binary merging, since it is more I/O- and cache-efficient. The high “branching factor“ allows to perform a considerable amount of work per item. This compensates for the costs of accessing the data on the (shared or external) memory.

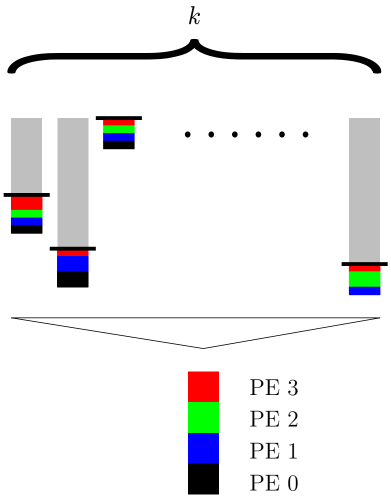


Figure 8.5: Schema of parallel multiway merging, in this case merging a specified number of elements only.

We parallelize multiway merging using multiway partitioning. Let $n = \sum_{i=1}^{k} |S_i|$. In parallel, each processing element (PE) $i > 1$ does multiway partitioning on the set of sequences, requesting global rank $\lceil \frac{i \cdot n}{p} \rceil$. Then, each PE i merges the part of the input that is determined by the locally computed splitting positions and by the splitting positions computed by PE $i + 1$. For $i = 0$ we use splitting local positions $(0, \ldots, 0)$. The destination offset is also clear from the ranks. Figure 8.5 shows a schematic view of the algorithm.

The loser tree data structure [48] keeps the next element of each sequence. The total execution time of the algorithm is $O(n/p \log k + k \log k \cdot \log \max_i |S_i|)$.

Based on the components explained so far, it is very easy to construct a parallel sorting algorithm. Each PE sorts about n/p elements sequentially, plus minus 1 due to the rounding. The resulting sorted sequences are merged using parallel multiway merging afterwards. We assume the sequential algorithm to run in time $O(n \log n)$, i. e. $O(n/p \log{(n/p)})$ per PE. Substituting $k = p$ and $S_i = n/p$ in the formula for multiway merging results in $O(n/p \log p + p \log p \cdot \log{(n/p)})$ The sequential sorting and the actual merging add up to $O(n/p \log n)$, but there is still the partitioning, so we have in total $O(n/p \log n + p \log p \cdot \log{(n/p)})$. Figure 8.6 illustrates the algorithm.

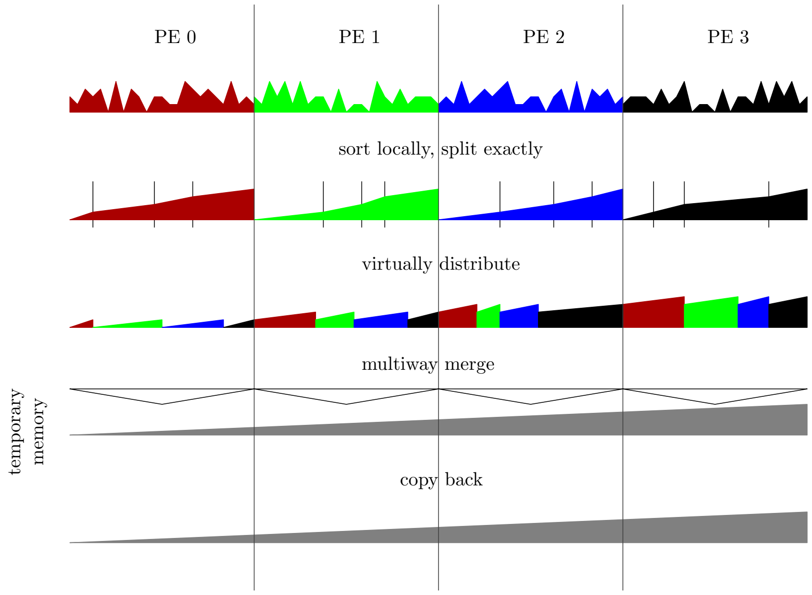


Figure 8.6: Schema of parallel multiway mergesort. For shared memory, the distribution depicted in the middle row takes place only virtually, i. e. the PEs read the data from where it is already, there is no copying. The final copying step is optional, only necessary if the result is expected in the same place as the input.

Chapter 9

String Algorithms

For reference, you are encouraged to consult one or more of the following text books:

- D. Gusfield: *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
- M. Crochemore, W. Rytter, *Jewels of Stringology*. World Scientific, 2002.
- M. Crochemore, C. Hancart, T. Lecroq, *Algorithms on Strings*. Cambridge UP, 2007.
- D. Adjeroh, T. Bell, and A. Mukherjee: *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching*. Springer, 2008.

9.1 Sorting Strings

Let S be a sequence of strings. Our task is to sort the elements in S *lexicographically*. We could take a general-purpose sorting algorithm for this task, e.g. merge-sort. However, such general-purpose algorithm take $O(n \lg n)$ *comparisons* to sort the date. Because in the case of strings such comparisons can be very expensive (e.g., comparing 2 long strings that are equal), we need a more specific algorithm. The code in Fig. 9.1 shows how to solve this task. It is a multi-key variant of the usual quicksort algorithm. Due to its three recursive calls, it is also called *ternary* quicksort.

```
Function mkqSort($S$ : Sequence of String, $i$ : $\mathbb{N}$) : Sequence of String
    assert $\forall e, e' \in S : e[1..i-1] = e'[1..i-1]$
    if $|S| \leq 1$ then return s                                    // base case
    pick $p \in S$ uniformly at random                               // pivot character
    return concatenation of
                  mkqSort($\langle e \in S : e[i] < p[i] \rangle, i$),
                  mkqSort($\langle e \in S : e[i] = p[i] \rangle, i+1$), and
                  mkqSort($\langle e \in S : e[i] > p[i] \rangle, i$)
```

Figure 9.1: Sorting strings lexicographically.

If the pivot p is chosen such that $p[i]$ is the *perfect* median among all $e[i]$'s, then the running time is $O(|S| \log |S| + \sum_{e \in S} |e|)$, which is optimal. With random selection of the median, this result holds in the *expected* case.

To be more precise, the running time can be bounded by $O(|S| \log |S| + d)$, where d denotes the sum of the *distinguishing prefixes* of all elements in S. This is because the set S will be of size 1 if all characters from the distinguishing prefix have been read, and the recursion stops in that case.

We saw in the tutorial that the partitioning into the three groups "<," "=," and ">" can be done *in-place*, which renders the concatenation of the arrays superfluous.

9.2 The Knuth-Morris-Pratt Algorithm

Let $P = p_1 p_2 \dots p_m$ be a pattern whose occurrences we want to find in a text $T = t_1 \dots t_n$ for $n \geq m$. The *naive* pattern-matching algorithm solves this task in $O(nm)$ time, as shown in Fig. 9.2. For the rest of this chapter, we use the notation $T_{i \dots j}$ as an abbreviation of $t_i t_{i+1} \dots t_j$.

We will now see how to change the naive algorithm such that it achieves optimal $O(n+m)$ time. The idea of the so-called *Knuth-Morris-Pratt* (or *KMP* for short) algorithm is as follows: in case of a mismatch between t_{i+j-1} and p_j, we shift P minimally such that a prefix of P is a suffix of the already matched part in T. To this end, we need the *border array* of P. For a string S of length ℓ, let $\alpha(S)$ denote the length of the longest prefix of $S_{1 \dots \ell-1}$ which is a suffix of $S_{2 \dots \ell}$. We can then define

$$\text{border}[j] = \begin{cases} -1 & \text{if } j = 1 \\ \alpha(P_{1 \dots j-1}) & \text{otherwise.} \end{cases}$$

```
Procedure naivePatternMatcher($P, T$)
    $i := 1$                                                    // index in $T$
    $j := 1$                                                    // index in $P$
    while $i \leq n - m + 1$
        while $j \leq m$ and $t_{i+j-1} = p_j$ do $j$++           // compare characters
        if $j > m$ then return "$P$ occurs at position $i$ in $T$"
        $i$++                                                   // advance in $T$
        $j := 1$                                                // restart
```

Figure 9.2: Naive pattern matching.

Here, we define $P_{i...j} = \epsilon$ (the empty string) if $j < i$, and we say that ϵ and P are both prefixes and suffixes of P. With this definition, it should be clear that it is safe to skip the next $j - \text{border}[j] - 2$ shifts of P, and immediately align P at position $i + j - \text{border}[j] - 1$ in T. Further, the first $\text{border}[j]$ characters of P need not be compared again, as they are guaranteed to match. See Fig. 9.3.

This results in the algorithm shown in Fig. 9.4.

The linear running time follows from the fact that all characters in T successfully compared with characters in P will never be compared again to any other part of P. Also, there are at most $n - m + 1$ unsuccessful character comparisons.

It remains to show how the border array is computed. Let us assume that $\text{border}[1..j - 1]$ has already been computed (we initialize $\text{border}[1]$ with -1.) In step j, we have to find the longest prefix of $P_{1...j-2}$ which is a suffix of $P_{2...j-1}$. Now if $p_{\text{border}[j-1]+1} = p_{j-1}$ then this prefix is $P_{1...\text{border}[j-1]+1}$, and we can hence set $\text{border}[j]$ to $\text{border}[j - 1] + 1$. (Observe that $\text{border}[j]$ cannot be longer, as otherwise $\text{border}[j - 1]$ should also be longer than it is, a contradiction to the the assumption that $\text{border}[j']$ is correct for all $j' < j$.) Otherwise ($p_{\text{border}[j-1]+1} \neq p_{j-1}$), we have to find the next shortest prefix of $P_{1...j-2}$ which is also a suffix

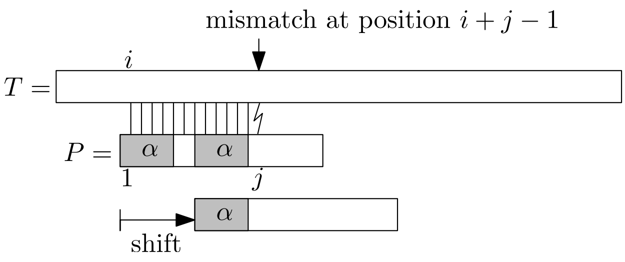


Figure 9.3: Idea of the KMP-algorithm. In this case, $\text{border}[j]$ is the length of α.

```
Procedure KMPPatternMatcher($P, T$)
    $i := 1$                                                    // index in $T$
    $j := 1$                                                    // index in $P$
    while $i \leq n - m + 1$
        while $j \leq m$ and $t_{i+j-1} = p_j$ do $j$++          // compare characters
        if $j > m$ then return "$P$ occurs at position $i$ in $T$"
        $i := i + j - \text{border}[j] - 1$                      // advance in $T$
        $j := \max\{1, border[j] + 1\}$         // skip first $\text{border}[j]$ characters of $P$
```

Figure 9.4: The Knuth-Morris-Pratt algorithm.

of $P_{2...j-1}$. But as it is shorter than $\text{border}[j-1]+1$, it must also be a suffix of $P_{2...\text{border}[j-1]+1}$. Hence, we can go to $\text{border}[\text{border}[j-1]+1]$. Then if $p_{\text{border}[\text{border}[j-1]+1]+1} = p_{j-1}$, we set $\text{border}[j]$ to $\text{border}[\text{border}[j-1]+1]+1$. This process continues until we find a match; we then set $\text{border}[j]$ accordingly. See also Fig. 9.5. The resulting algorithm is shown in Fig. 9.6

The linear running time follows from the fact that all entries in the border array that are "traversed" (skipped over) in the while-loop will never be considered again (due to the way $\text{border}[j]$ is set). Hence, the total number of executions of the while-loop is at most $2m$, which implies a linear overall running time.

Theorem 22
The Knuth-Morris-Pratt algorithm finds all occurences of a pattern of length m in a text of length n in overall $O(n+m)$ time, using additional $O(m)$ space.

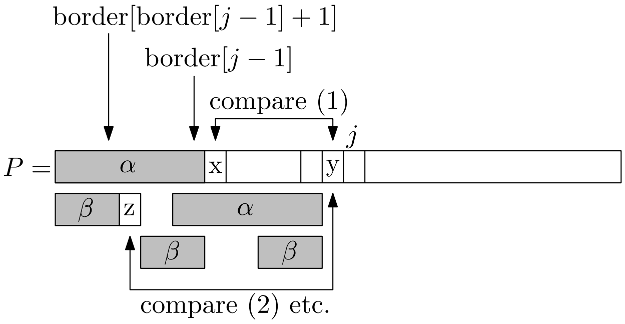


Figure 9.5: Sketch for computing the border array.

```
Procedure computeBorderArray($P$)
    border[1] := $-1$
    $i$ := border[1]                                        // position in $P$
    for $j = 2, \dots, m$
        while $i \geq 0$ and $p_{i+1} \neq p_{j-1}$ do $i$ = border[$i + 1$]
        $i$++
        border[$j$] := $i$
```

Figure 9.6: Algorithm for computing the border array.

9.3 Suffix Trees and Arrays

9.3.1 Suffix Trees

In this section we will introduce suffix trees, which, among many other things, can be used to solve the string matching task (find pattern P of length m in a text T of length n in $O(n + m)$ time). We already saw that other methods (Knuth-Morris-Pratt, e.g.) solve this task in the same time. So why do we need suffix trees?

The advantage of suffix trees over the other string-matching algorithms is that suffix trees are an *index* of the text. So, if T is *static* and there are several patterns to be matched against T, the $O(n)$-task for building the index needs to be done only once, and subsequent matching-tasks can be done in $O(m)$ time. If $m \ll n$, this is a clear advantage over the other algorithms.

Throughout this section, let $T = t_1 t_2 \dots t_n$ be a text over an alphabet Σ of size $|\Sigma| =: \sigma$. For $i \in \{1, 2, \dots, n\}$, $t_i t_{i+1} \dots t_n$ is called the *i-th suffix* of T and is denoted by $S_i := T_{i \dots n}$.

Definition 7
A compact Σ^+-tree *is a rooted tree $\mathcal{T} = (V, E)$ with edge labels from Σ^+ that fulfills the following two constraints:*

- *$\forall v \in V$: all outgoing edges from v start with a different $a \in \Sigma$.*
- *Apart from the root, all nodes have out-degree $\neq 1$.*

Definition 8
Let $\mathcal{T} = (V, E)$ be a compact Σ^+-tree.

- *For $v \in V$, $\overline{v}$ denotes the concatenation of all path labels from the root of $\mathcal{T}$ to v.*

- $|\overline{v}|$ *is called the* string-depth *of* v *and is denoted by* $d(v)$.
- $\mathcal{T}$ *is said to* display $\alpha \in \Sigma^*$ *iff* $\exists v \in V, \beta \in \Sigma^* : \overline{v} = \alpha\beta$.
- *If* $\overline{v} = \alpha$ *for* $v \in V, \alpha \in \Sigma^*$, *we also write* $\overline{\alpha}$ *to denote* v.
- *words*$(\mathcal{T})$ *denotes all strings in* Σ^* *that are displayed by* $\mathcal{T}$: *words*$(\mathcal{T}) = \{\alpha \in \Sigma^* : \mathcal{T}$ *displays* $\alpha\}$

We are now ready to define suffix trees.

Definition 9
Let factor(T) *denote the set of all factors of* T, *factor*$(T) = \{T_{i\ldots j} : 1 \leq i \leq j \leq n\}$. *The* suffix tree *of* T *is a compact* Σ^+*-tree* $\mathcal{S}$ *with words*$(\mathcal{S}) =$ *factor*(T).

For several reasons, we shall find it useful that each suffix ends in a leaf of $\mathcal{S}$. This can be accomplished by adding a new character $\$ \notin \Sigma$ to the end of T, and build the suffix tree over $T\$$.

From now on, we assume that T terminates with a \$, and we define \$ to be lexicographically smaller than all other characters in Σ: $\$ < a$ for all $a \in \Sigma$. This gives a one-to-one correspondence between T's suffixes and the leaves of $\mathcal{S}$, which implies that we can *label the leaves* with a function l by the start index of the suffix they represent: $l(v) = i \iff \overline{v} = S_i$. This also explains the name "suffix tree."

Remark: The outgoing edges at internal nodes v of the suffix tree can be implemented in two fundamentally different ways:

1. as arrays of size σ
2. as arrays of size s_v, where s_v denotes the number of v's children

Approach (1) has the advantage that the outgoing edge whose edge label starts with $\alpha \in \Sigma$ can be located in $O(1)$ time, but the complete suffix tree uses space $O(n\sigma)$, which can be as bad as $O(n^2)$. Hence, we assume that approach (2) is used, which implies that locating the correct outgoing edge takes $O(\log \sigma)$ time (using binary search). Note that the space consumption of approach (2) is always $O(n)$, independent of σ.

9.3.2 Searching in Suffix Trees

Let P be a pattern of length m. We will be concerned with the two following problems:

Counting: Return the number of matches of P in T. Formally, return the *size* of $O_P = \{i \in [1, n] : T_{i\ldots i+m-1} = P\}$

Reporting: Return all occurrences of P in T, i.e., return the set O_P.

With suffix trees, the *counting-problem* can be solved in $O(m \log \sigma)$ time: traverse the tree from the root downwards, in each step locating the correct outgoing edge, until P has been scanned completely. More formally, suppose that $P_{1\dots i-1}$ has already been parsed for some $1 \leq i < m$, and our position in the suffix tree $\mathcal{S}$ is at node v ($\overline{v} = P_{1\dots i-1}$). We then find v's outgoing edge e whose label starts with P_i. This takes $O(\log \sigma)$ time. We then compare the label of e character-by-character with $P_{i\dots m}$, until we have read all of P ($i = m$), or until we have reached position $j \geq i$ for which $\overline{P_{1\dots j}}$ is a node v' in $\mathcal{S}$, in which case we continue the procedure at v'. This takes a total of $O(m \log \sigma)$ time. Suppose the search procedure has brought us successfully to a node v, or to the incoming edge of node v. We then output the size of $\mathcal{S}_v$, the subtree of $\mathcal{S}$ rooted at v. This can be done in constant time, assuming that we have labeled all nodes in $\mathcal{S}$ with their subtree sizes. This answers the *counting query*. For the *reporting query*, we output the labels of all leaves in $\mathcal{S}_v$ (recall that the leaves are labeled with text positions).

Theorem 23
The suffix tree allows to answer counting queries in $O(m \log \sigma)$ time, and reporting queries in $O(m \log \sigma + |O_P|)$ time.

An important *implementation detail* is that the edge labels in a suffix tree are represented by a pair (i, j) of *integers*, $1 \leq i \leq j \leq n$, such that $T_{i\dots j}$ is equal to the corresponding edge label. This ensures that an edge label uses only a constant amount of memory.

From this implementation detail and the fact that $\mathcal{S}$ contains exactly n leaves and hence less than n internal nodes, we can formulate the following theorem:

Theorem 24
A suffix tree occupies $O(n)$ space in memory.

9.3.3 Suffix- and LCP-Arrays

We will now introduce two arrays that are closely related to the suffix tree, the *suffix array* SA and the *LCP-array* LCP.

Definition 10
The suffix array SA *of T is a permutation of $\{1, 2, \dots, n\}$ such that* $\mathsf{SA}[i]$ *is the i-th smallest suffix in lexicographic order: $S_{\mathsf{SA}[i-1]} < S_{\mathsf{SA}[i]}$ for all $1 < i \leq n$.*

The following observation relates the suffix array SA with the suffix tree S.

Remark 25
If we do a lexicographically-driven depth-first search through $\mathcal{S}$ (visit the children in lexicographic order of the first character of their corresponding edge-label), then the leaf-labels seen in this order give the suffix-array SA.

The second array LCP builds on the suffix array:

Definition 11
The LCP-array LCP *of T is defined such that* $\mathsf{LCP}[1] = 0$, *and for all $i > 1$,* $\mathsf{LCP}[i]$ *holds the length of the longest common prefix of $S_{\mathsf{SA}[i]}$ and $S_{\mathsf{SA}[i-1]}$.*

To relate the LCP-array LCP with the suffix tree $\mathcal{S}$, we need to define the concept of lowest common ancestors:

Definition 12
Given a tree $\mathcal{T} = (V, E)$ and two nodes $v, w \in V$, the lowest common ancestor *of v and w is the deepest node in $\mathcal{T}$ that is an ancestor of both v and w. This node is denoted by $\textsc{lca}(v, w)$.*

Remark 26
The string-depth of the lowest common ancestor of the leaves labeled $\mathsf{SA}[i]$ *and* $\mathsf{SA}[i-1]$ *is given by the corresponding entry* $\mathsf{LCP}[i]$ *of the LCP-array, in symbols:* $\forall i > 1 : \mathsf{LCP}[i] = d(\textsc{lca}(\overline{S_{\mathsf{SA}[i]}}, \overline{S_{\mathsf{SA}[i-1]}}))$.

9.3.4 Searching in Suffix Arrays

We can use a *plain suffix array* SA to search for a pattern P, using the ideas of *binary search*, since the suffixes in SA are *sorted* lexicographically and hence the occurrences of P in T form an *interval* in SA. The algorithm below performs two binary searches. The first search locates the starting position s of P's interval in SA, and the second search determines the end position r. A *counting query* returns $r - s + 1$, and a *reporting* query returns the numbers $\mathsf{SA}[s], \mathsf{SA}[s+1], \ldots, \mathsf{SA}[r]$.

Note that both while-loops the algorithm above make sure that either l is increased or r is decreased, so they are both guaranteed to terminate. In fact, in the first while-loop, r always points one position *behind* the current search interval, and r is *decreased* in case of equality (when $P = T_{\mathsf{SA}[q] \ldots \min\{\mathsf{SA}[q]+m-1, n\}}$). This makes sure that the first while-loop finds the *leftmost* position of P in SA. The second loop works symmetrically. Note further that in the second while-loop it is enough to check for lexicographical equality, as the whole search is done in the interval of SA where all suffixes are lexicographically no less than P.

Theorem 27
The suffix array allows to answer counting queries in $O(m \log n)$ time, and reporting queries in $O(m \log n + |O_P|)$ time.

```
Procedure SAsearch($P_{1...m}$)
    $l := 1; r := n + 1$
    while $l < r$ do    //search left index
        $q := \lfloor \frac{l+r}{2} \rfloor$
        if $P >_{\text{lex}} T_{\mathsf{SA}[q]\ldots\min\{\mathsf{SA}[q]+m-1,n\}}$
            then $l := q + 1$
            else $r := q$
    $s := l; l{-}{-}; r := n$
    while $l < r$ do    //search right index
        $q := \lceil \frac{l+r}{2} \rceil$
        if $P =_{\text{lex}} T_{\mathsf{SA}[q]\ldots\min\{\mathsf{SA}[q]+m-1,n\}}$
            then $l := q$
            else $r := q - 1$
    return $[s, l]$
```

Figure 9.7: The binary search algorithm for suffix arrays.

9.3.5 Construction of Suffix Trees from Suffix- and LCP-Arrays

Assume for now that we are given T, SA, and LCP, and we wish to construct $\mathcal{S}$, the suffix tree of T. We will show in this section how to do this in $O(n)$ time. Later, we will also see how to construct SA and LCP only from T in linear time. In total, this will give us an $O(n)$-time construction algorithm for suffix trees.

The idea of the algorithm is to insert the suffixes into $\mathcal{S}$ in the order of the suffix array: $S_{\mathsf{SA}[1]}, S_{\mathsf{SA}[2]}, \ldots, S_{\mathsf{SA}[n]}$. To this end, let $\mathcal{S}_i$ denote the partial suffix tree for $0 \leq i \leq n$ ($\mathcal{S}_i$ is the compact Σ^+-tree with *words*$(\mathcal{S}_i) = \{T_{\mathsf{SA}[k]\ldots j} : 1 \leq k \leq i, \mathsf{SA}[k] \leq j \leq n\}$). In the end, we will have $\mathcal{S} = \mathcal{S}_n$.

We start with $\mathcal{S}_0$, the tree consisting only of the root (and thus displaying only ϵ). In step $i + 1$, we climb up the *rightmost path* of $\mathcal{S}_i$ (i.e., the path from the leaf labeled $\mathsf{SA}[i]$ to the root) until we meet the deepest node v with $d(v) \leq \mathsf{LCP}[i + 1]$. If $d(v) = \mathsf{LCP}[i + 1]$, we simply insert a new leaf x to $\mathcal{S}_i$ as a child of v, and label (v, x) by $T_{\mathsf{SA}[i+1]+\mathsf{LCP}[i+1]\ldots n}$. Leaf x is labeled by $\mathsf{SA}[i + 1]$. This gives us $\mathcal{S}_{i+1}$.

Otherwise (i.e., $d(v) < \mathsf{LCP}[i + 1]$), let w be the child of v on $\mathcal{S}_i$'s rightmost path. In order to obtain $\mathcal{S}_{i+1}$, we *split up* the edge (v, w) as follows.

1. Delete (v, w).

2. Add a new node y and a new edge (v, y). (v, y) gets labeled by $T_{\mathsf{SA}[i]+d(v)\ldots\mathsf{SA}[i]+\mathsf{LCP}[i+1]-1}$.

3. Add (y, w) and label it by $T_{\mathsf{SA}[i]+\mathsf{LCP}[i+1]\ldots\mathsf{SA}[i]+d(w)-1}$.

4. Add a new leaf x (labeled $\mathsf{SA}[i+1]$) and an edge (y, x). Label (y, x) by $T_{\mathsf{SA}[i+1]+\mathsf{LCP}[i+1]\ldots n}$.

The correctness of this algorithm follows from observations 25 and 26 above. Let us now consider the execution time of this algorithm. Although climbing up the rightmost path could take $O(n)$ time in a single step, a simple amortized argument shows that the running time of this algorithm can be bounded by $O(n)$ in total: each node traversed in step i (apart from the last) is *removed* from the rightmost path and will not be traversed again for all subsequent steps $j > i$. Hence, at most $2n$ nodes are traversed in total.

Theorem 28
We can construct T's suffix tree in linear time from T's suffix- and LCP-array.

9.3.6 Linear-Time Construction of Suffix Arrays

This section describes the DC3-algorithm for constructing suffix arrays (Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt: Linear Work Suffix Array Construction. *Journal of the ACM* **53**: 1–19, 2005). For an easier presentation, we assume in this section that T is index from 0 to $n-1$, $T = t_0 t_1 \ldots t_{n-1}$.

Definition 13
If $x \bmod y = z$, we write $x \equiv z (\bmod y)$ or simply $x \equiv z(y)$.

Our general approach will be *recursive*: first construct the suffix array SA_{12} for the suffixes S_i with $i \not\equiv 0(3)$ by a recursive call to the suffix sorting routine. From this, derive the suffix array SA_0 for the suffixes S_i with $i \equiv 0(3)$. Then merge SA_{12} and SA_0 to obtain the final suffix array.

Creation of SA_{12} by Recursion

In order to call the suffix sorting routine recursively, we need to construct a new text T' from whose suffix array SA' we can derive SA_{12}. To this end, we look at all *character triplets* $t_i t_{i+1} t_{i+2}$ with $i \not\equiv 0(3)$, and sort the resulting set of triplets $S = \{t_i t_{i+1} t_{i+2} : i \not\equiv 0(3)\}$ with a bucket-sort in $O(n)$ time. (To have all triplets well-defined, we pad T with sufficiently many \$'s at the end.)

We define T' as follows: its first half consists of the bucket-numbers of $t_i t_{i+1} t_{i+2}$ for increasing $i \equiv 1(3)$, and its second half consists of the same for the triplets with $i \equiv 2(3)$.

We now build the suffix array SA' for T' by a recursive call (stop the recursion if $|T'| = O(1)$). This already gives us sorting of the suffixes starting at positions $i \not\equiv 0(3)$, because of the following:

- The suffixes starting at $i \equiv 2(3)$ in T have a one-to-one correspondence to the suffixes in T' and are hence in correct lexicographic order.
- The suffixes starting at $i \equiv 1(3)$ in T are longer than they should be (because of the bucket numbers of the triplets starting at $2(3)$), but due to the \$'s in the middle of T' this "tail" does not influence the result.

Thus, all that remains to be done is to convert the indices in T' to indices in T, which is done as follows:

$$\mathsf{SA}_{12}[i] = \begin{cases} 1 + 3\mathsf{SA}'[i] & \text{if } \mathsf{SA}'[i] < \lceil \frac{|T'|}{2} \rceil \\ 2 + 3(\mathsf{SA}'[i] - \lceil \frac{|T'|}{2} \rceil) & \text{otherwise} \end{cases}$$

Creation of SA_0

The following lemma follows immediately from the definition of SA_{12}.

Lemma 29
Let $i, j \equiv 0(3)$. Then $S_i < S_j$ iff $t_i < t_j$, or $t_i = t_j$ and $i + 1$ appears before $j + 1$ in SA_{12}.

This suggests the following strategy to construct SA_0:

1. Initialize SA_0 with all numbers $0 \leq i < n$ for which $i \equiv 0(3)$.
2. Bucket-sort SA_0, where $\mathsf{SA}_0[i]$ has sort-key $t_{\mathsf{SA}_0[i]}$.
3. In a left-to-right scan of SA_{12}: if $\mathsf{SA}_{12}[i] \equiv 1(3)$, move $\mathsf{SA}_{12}[i] - 1$ to the current beginning of its bucket.

Merging SA_{12} with SA_0

We scan SA_0 and SA_{12} simultaneously. The suffixes from SA_0 and SA_{12} can be compared among each other in $O(1)$ time by the following lemma, which follows again directly from the definition of SA_{12}.

Lemma 30
Let $i \equiv 0(3)$.

1. *If $j \equiv 1(3)$, then $S_i < S_j$ iff $t_i < t_j$, or $t_i = t_j$ and $i + 1$ appears before $j + 1$ in SA_{12}.*
2. *If $j \equiv 2(3)$, then $S_i < S_j$ iff $t_i < t_j$, or $t_i = t_j$ and $t_{i+1} < t_{j+1}$, or $t_i t_{i+1} = t_j t_{j+1}$ and $i + 2$ appears before $j + 2$ in SA_{12}.*

One should note that in both of the above cases the values $i+1$ and $j+1$ (or $i+2$ and $j+2$, respectively) appear in SA_{12} — this is why it is enough to compare at most 2 characters before one can derive the lexicographic order of S_i and S_j from SA_{12}.

In order to check efficiently whether i appears before j in SA_{12}, we can use the *inverse suffix array* SA_{12}^{-1} of SA_{12}, defined by

$$\mathsf{SA}_{12}^{-1}[\mathsf{SA}_{12}[i]] = i$$

for all i. With this, it is easy to see that i appears before j in SA_{12} iff $\mathsf{SA}_{12}^{-1}[i] < \mathsf{SA}_{12}^{-1}[j]$.

The running time $T(n)$ of the whole suffix sorting algorithm presented in this section is given by the recursion $T(n) = \mathcal{T}(2n/3) + O(n)$, which solves to $T(n) = O(n)$.

Theorem 31
We can construct the suffix array for a text of length n in $O(n)$ time.

9.3.7 Linear-Time Construction of LCP-Arrays

It remains to be shown how the LCP-array LCP can be constructed in $O(n)$ time. Here, we assume that we are given T, SA, and SA^{-1}, the latter being the inverse suffix array.

We will construct LCP *in the order of the inverse suffix array* (i.e., filling $\mathsf{LCP}[\mathsf{SA}^{-1}[i]]$ before $\mathsf{LCP}[\mathsf{SA}^{-1}[i+1]]$), because in this case we know that LCP cannot decrease too much, as shown next (see also Fig. 9.8 for what follows).

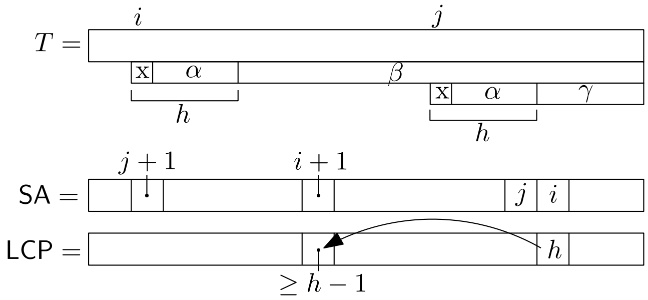


Figure 9.8: Sketch to the proof of Lemma 32. $S_i = \mathrm{x}\alpha\beta$ and $S_j = \mathrm{x}\alpha\gamma$.

Going from suffix S_i to S_{i+1}, we see that the latter equals the former, but with the first character t_i truncated. Let $h = \mathsf{LCP}[i]$. Then the suffix S_j, $j = \mathsf{SA}[\mathsf{SA}^{-1}[i] - 1]$, has a longest common prefix with S_i of length h. So S_{i+1} has a longest common prefix with S_{j+1} of length $h-1$. But every suffix S_k that is lexicographically between S_{j+1} and S_{i+1} must have a longest common prefix with S_{j+1} that is at least $h-1$ characters long (for otherwise S_k would not be in lexicographic order). In particular, the suffix right before S_{i+1} in SA, which is suffix $S_{\mathsf{SA}[\mathsf{SA}^{-1}[i+1]-1]}$, must share a common prefix with S_{i+1} of length at least $h-1$. Hence, $\mathsf{LCP}[\mathsf{SA}^{-1}[i+1]] \geq h-1$. We have thus proved the following:

Lemma 32
For all $1 \leq i < n$: $\mathsf{LCP}[\mathsf{SA}^{-1}[i+1]] \geq \mathsf{LCP}[\mathsf{SA}^{-1}[i]] - 1$.

This gives rise to the following elegant algorithm to construct LCP:

```
Procedure ComputeLCP
    $h := 0, \mathsf{LCP}[1] := 0$
    for $i = 1, \ldots, n$ do
        if $\mathsf{SA}^{-1}[i] \neq 1$ then
            while $t_{i+h} = t_{\mathsf{SA}[\mathsf{SA}^{-1}[i]-1]+h}$ do $h++$
            $\mathsf{LCP}[\mathsf{SA}^{-1}[i]] := h$
            $h := \max(0, h-1)$
```

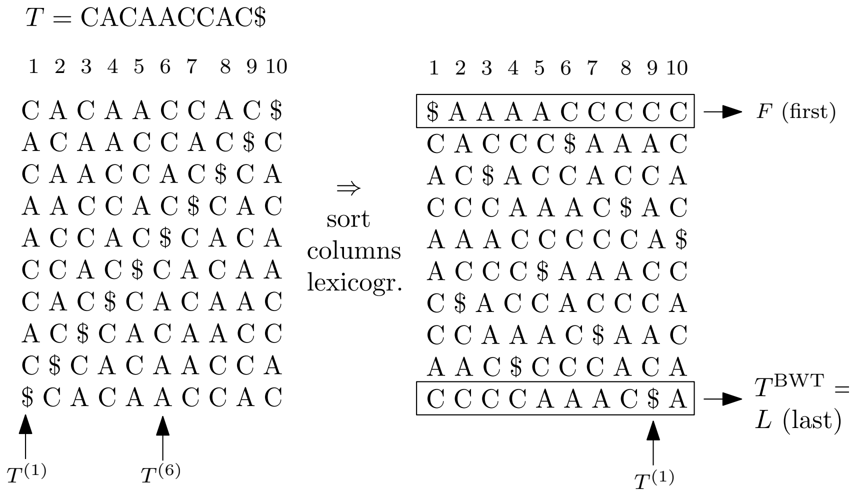
Figure 9.9: Constructing the LCP-array in linear time.

The linear running time follows because h starts and ends at 0, is always less than n and decreased at most n times in the last line. Hence, the number of times where hna is increased in line 5 is bounded by n, so there are at most $2n$ character comparisons in the whole algorithm. We have proved:

Theorem 33
The LCP-array for a text of length n *can be constructed in* $O(n)$ *time.*

9.4 The Burrows-Wheeler Transformation

The Burrows-Wheeler Transformation was originally invented for text *compression*. Nonetheless, it was noted soon that it is also a very useful tool in text *indexing*. In this section, we introduce the transformation and briefly review its merits for compression.



```
T = CACAACCAC$

1 2 3 4 5 6 7 8 9 10                    1 2 3 4 5 6 7 8 9 10

C A C A A C C A C $                     $ A A A A C C C C C   → F (first)
A C A A C C A C $ C                     C A C C C $ A A A C
C A A C C A C $ C A        ⇒            A C $ A C C A C C A
A A C C A C $ C A C       sort          C C C A A A C $ A C
A C C A C $ C A C A      columns        A A A C C C C C A $
C C A C $ C A C A A      lexicogr.      A C C C $ A A A C C
C A C $ C A C A A C                     C $ A C C A C C C A
A C $ C A C A A C C                     C C A A A C $ A A C
C $ C A C A A C C A                     A A C $ C C C A C A
$ C A C A A C C A C                     C C C C A A A C $ A   → T^BWT = L (last)
↑         ↑                                               ↑
T(1)      T(6)                                            T(1)
```

Figure 9.10: Example of the Borrows-Wheeler-Transformation

9.4.1 The Transformation

Definition 14
*Let $T_{1 \ldots n} = t_1 \ldots t_n$ be a text of length n, where $t_n = \$$ is a unique character lexicographically smaller than all other characters in Σ. Then the i-*th cyclic shift *of T is $T_{i \ldots n} T_{1 \ldots i-1}$. We denote it by $T^{(i)}$.*

The *Burrows-Wheeler-Transformation* (BWT) is obtained by the following steps:

1. Write all cyclic shifts $T^{(i)}$, $1 \leq i \leq n$, column-wise next to each other.

2. Sort the columns lexicographically.

3. Output the last row. This is T^{BWT}.

The text T^{BWT} in the last row is also denoted by L (last), and the text in the first row by F (first). Note:

- Every row in the BWT-matrix is a permutation of the characters in T.

- Row F is a sorted list of all characters in T.

- In row $L = T^{\text{BWT}}$, similar characters are grouped together. This is why T^{BWT}can be compressed more easily than T.

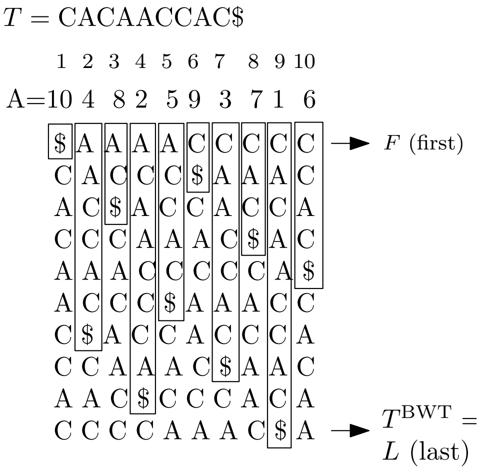


Figure 9.11: Correspondence between BWT and the suffix array.

9.4.2 Construction of the BWT

The BWT-matrix needs *not* to be constructed *explicitly* in order to obtain T^{BWT}. Since T is terminated with the special character \$, which is lexicographically smaller than any $a \in \Sigma$, the shifts $T^{(i)}$ are sorted exactly like T's suffixes. Because the last row consists of the characters *preceding* the corresponding suffixes, we have

$$t_i^{\mathrm{BWT}} = L[i] = t_{\mathsf{SA}[i]-1} (= t_n^{(\mathsf{SA}[i])}) ,$$

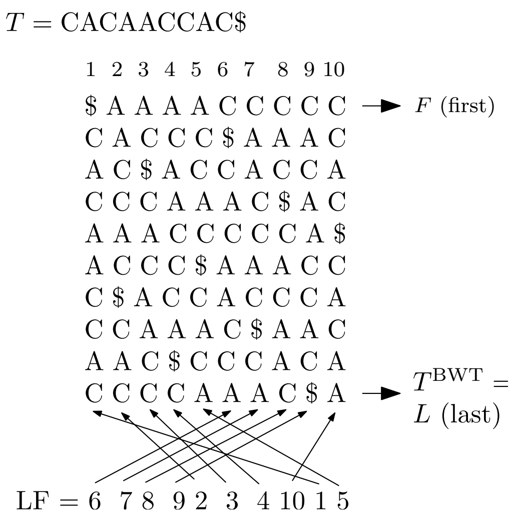
where SA denotes again T's suffix array, and t_0 is defined to be t_n (read T cyclically!). See also Fig. 9.11. Because the suffix array can be constructed in linear time (Thm. 31), we get:

Theorem 34
The BWT of a text length-n text over an integer alphabet can be constructed in $O(n)$ time.

9.4.3 The Reverse Transformation

The amazing property of the BWT is that it is not a random permutation of T's letters, but that it can be *transformed back* to the original text T. For this, we need the following definition:

Definition 15
Let F and L be the strings resulting from the BWT. *Then the* last-to-front *mapping*



$T = \text{CACAACCAC\$}$

```
1 2 3 4 5 6 7 8 9 10
$ A A A A C C C C C  → F (first)
C A C C C $ A A A C
A C $ A C C A C C A
C C C A A A C $ A C
A A A C C C C C A $
A C C C $ A A A C C
C $ A C C A C C C A
C C A A A C $ A A C
A A C $ C C C A C A
C C C C A A A C $ A  → T^BWT = L (last)

LF = 6 7 8 9 2 3 4 10 1 5
```

Figure 9.12: Example of the LF-mapping.

LF *is a function* LF : $[1, n] \to [1, n]$, *defined by*

$$\text{LF}(i) = j \iff T^{(\mathsf{SA}[j])} = (T^{(\mathsf{SA}[i])})^{(n)} \; (\iff \mathsf{SA}[j] = \mathsf{SA}[i] + 1) \ .$$

(Remember that $T^{(\mathsf{SA}[i])}$ *is the* i*'th column in the* BWT*-matrix, and* $(T^{(\mathsf{SA}[i])})^{(n)}$ *is that column rotated by one character downwards.)*

Thus, LF(i) tells us the position in F where $L[i]$ occurs. See Fig. 9.12 for an example.

Remark 35
Equal characters preserve the same order in F *and* L. *That is, if* $L[i] = L[j]$ *and* $i < j$, *then* LF$(i) <$ LF(j). *To see why this is so, recall that the* BWT*-matrix is sorted lexicographically. Because both the* LF(i)*'th and the* LF(j)*'th column start with the same character* $a = L[i] = L[j]$, *they must be sorted according to what follows this character* a, *say* α *and* β. *But since* $i < j$, *we know* $\alpha <_{\mathit{lex}} \beta$, *hence* LF$(i) <$ LF(j). *See also Fig. 9.13.*

This observation allows us to compute the LF-mapping *without knowing the suffix array* of T.

Definition 16
Let T *be a text of length* n *over an alphabet* Σ, *and let* $L = T^{\text{BWT}}$ *be its* BWT.

- *Define* $C : \Sigma \to [1, n]$ *such that* $C(a)$ *is the number of occurrences in* T *of characters that are lexicographically smaller than* $a \in \Sigma$.

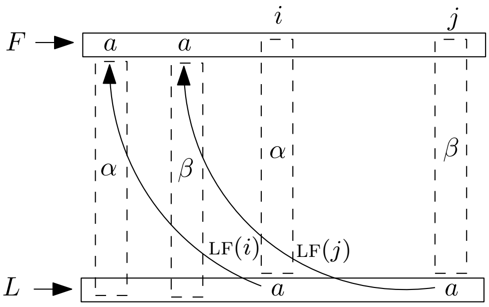


Figure 9.13: Sketch to observation 35.

- *Define* $\textsc{occ} : [1,n] \rightarrow [1,n]$ *such that* $\textsc{occ}(i)$ *is the number of occurrences of* $L[i]$ *in* L*'s length-*i*-prefix* $L[1,i]$.

Lemma 36
With the definitions above,

$$\textsc{lf}(i) = C(L[i]) + \textsc{occ}(i) \ .$$

This gives rise to the following algorithm to recover T from $L = T^{\textsc{bwt}}$ (see Fig. 9.14 for an example).

1. Scan $L = T^{\textsc{bwt}}$ and compute array $C[1,\sigma]$.

2. Compute the first row F from C; as F consists of all characters in L sorted lexicographically, this step is trivial.

3. Compute $\textsc{occ}(i)$ for all $1 \leq i \leq n$.

4. Recover T *from right to left*: we know that $t_n = \$$, and the corresponding cyclic shift $T^{(n)}$ appears in column 1 in $\textsc{bwt}$. Hence, $t_{n-1} = L[1]$. Shift $T^{(n-1)}$ appears in column $\textsc{lf}(1)$, and thus $t_{n-2} = L[\textsc{lf}(1)]$. This continues until the whole text has been recovered:

$$t_{n-i} = L[\underbrace{\textsc{lf}(\textsc{lf}(\ldots(\textsc{lf}(1))\ldots))}_{i-1 \text{ applications of } \textsc{lf}}]$$

$$C = \overset{\$}{0}\ \overset{\mathrm{A}}{1}\ \overset{\mathrm{C}}{5}$$

$$F = \$\ \mathrm{A}\ \mathrm{A}\ \mathrm{A}\ \mathrm{A}\ \mathrm{C}\ \mathrm{C}\ \mathrm{C}\ \mathrm{C}\ \mathrm{C}$$

$$L = \mathrm{C}\ \mathrm{C}\ \mathrm{C}\ \mathrm{C}\ \mathrm{A}\ \mathrm{A}\ \mathrm{A}\ \mathrm{C}\ \$\ \mathrm{A}$$

$$\textsc{occ} = 1\ 2\ 3\ 4\ 1\ 2\ 3\ 5\ 1\ 4$$

$$\begin{array}{r|c|l}
T_n = & \$ & , k = 1 \\
L[1] = \mathrm{C} \Rightarrow T_{n-1} = & \mathrm{C} & , k = \mathrm{LF}(1) = 6 \\
L[6] = \mathrm{A} \Rightarrow T_{n-2} = & \mathrm{A} & , k = \mathrm{LF}(6) = 3 \\
L[3] = \mathrm{C} \Rightarrow T_{n-3} = & \mathrm{C} & , k = \mathrm{LF}(3) = 8 \\
L[8] = \mathrm{C} \Rightarrow T_{n-4} = & \mathrm{C} & , k = \mathrm{LF}(8) = 10 \\
L[10] \text{ etc.} & &
\end{array}$$

↑

T reversed

Figure 9.14: Recovering the original text from the BWT.

9.4.4 Compression

Storing T^{BWT} *plainly* needs the same space as storing the original text T. However, because equal characters are grouped together in T^{BWT}, we can compress T^{BWT} in a second stage. Here, we show how this can be achieved by a combination of Move-to-Front (MTF) and Huffman Coding.

1. Initialize a list Y containing each character in Σ in alphabetic order.
2. In a left-to-right scan of T^{BWT}, $(i = 1, \ldots, n)$, compute a new array $R[1, n]$ (see Fig. 9.15 for an example):
   - Write the position of character t_i^{BWT} in Y to $R[i]$.
   - Move character t_i^{BWT} to the front of Y.
3. Encode the resulting string/array R with any kind of reversible compressor, e. g. Huffman, into a string C (see Fig. 9.16 for an example).

Remark 37
MTF produces "many small" numbers for equal characters that are "close together" in T^{BWT}. These can be compressed using an order-0 compressor, e. g. Huffman.

Both steps (Huffman & MTF) are easy to reverse.

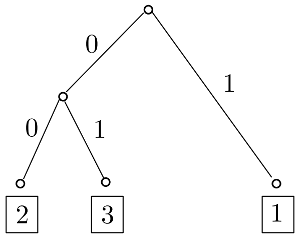
T^{BWT} =C C C C A A A C $ A

i	Y_{old}	T_i^{BWT}	$R[i]$	Y_{new}
1	$AC	C	3	C$A
2	C$A	C	1	C$A
3	C$A	C	1	C$A
4	C$A	C	1	C$A
5	C$A	A	3	AC$
6	AC$	A	1	AC$
7	AC$	A	1	AC$
8	AC$	C	2	CA$
9	CA$	$	3	$CA
10	$CA	A	3	A$C

Figure 9.15: The Move-to-Front algorithm.

character	frequency
1	5
2	1
3	4

⇓



R[i]
3
1
1
1
3
1
1
2
3
3

⇓

C = 011110111000101

Figure 9.16: Huffman Encoding.

9.5 Lempel-Ziv Compression

There is also a different (easier) way to compress a string: Lempel-Ziv compression or LZ for short. The idea of this algorithm is to divide the string $S = s_1 \dots s_n$ into *phrases* of previously seen substrings. Those previously seen substrings constitute the *dictionary*. To achieve compression, instead of storing those phrases plainly, we just store *references* to positions in the dictionary.

More precisely, suppose we have already parsed a prefix $s_1 s_2 \dots s_{j-1}$ into phrases $p_1 p_2 \dots p_{m-1}$, with $p_\ell \in D$ for all $1 \le \ell < m$ and $D \subseteq \Sigma^\star$ being the current dictionary. The next phrase p_m is then defined as the longest prefix p of $s_j s_{j+1} \dots s_n$, say $p = s_j \dots s_{i-1}$, such that $p \in D$. To keep the phrases getting longer (and hence compress S), we then add $p \cdot s_i$ to the dictionary D, where "$\cdot$" denotes concatenation of strings. To get the process started, we initialze the dictionary D with all letters from the alphabet Σ. The complete algorithm can be seen in Fig. 9.17.

```
Procedure naiveLZCompress($\langle s_1, \dots, s_n \rangle, \Sigma$)
    $D$:= $\Sigma$                                  // Init Dictionary
    $p$:= $s_1$                                     // current string
    for $i$ := 2 to $n$ do
        if $p \cdot s_i \in D$ then $p$:= $p \cdot s_i$
        else
            output code for $p$
            $D$:= $D \cup p \cdot s_i$
            $p$:= $s_i$
    output code for $p$
```

Figure 9.17: LZ-compression.

An example is shown in Fig. 9.18. There, the last column shows the current dictionary, with the numbers of the entries in brackets. The column labeled 'output' is the compressed string S. It consists of numbers of entries in the current dictionary.

Decompression reverses the process of compression. By the way we parsed the string (using only previously seen substrings as new phrases), then if we add any decoded phrase to the dictionary we can be sure that all codes we see have already an entry in the dictionary — almost! The only exception occurs when during the compression we have $p_m = p_{m-1} \cdot a$ for some $a \in \Sigma$. But then after the parsing of p_{m-1} we have inserted $p_{m-1} \cdot p_m[1]$ into D. Now, because p_{m-1} is a non-empty

i	p	output	s_i	$D\cup =$
1	$\perp$	-	a	a(1),b(2),c(3),d(4),r(5)
2	a	1	b	ab(6)
3	b	2	r	br(7)
4	r	5	a	ra(8)
5	a	1	c	ac(9)
6	c	3	a	ca(10)
7	a	1	d	ad(11)
8	d	4	a	da(12)
9	a	-	b	-
10	ab	6	r	abr(13)
11	r	-	a	-
-	ra	8	-	-

Figure 9.18: Example of LZ-compression.

prefix of p_m, $p_m[1] = p_{m-1}[1]$, so we can deduce that the unknown character a is the first letter of p_{m-1}.

The resulting algorithm is shown in Fig. 9.19, where the above "special case" is taken care of by updating the dictionary *before* outputting the decoded phrase.

```
Procedure naiveLZDecode($\langle c_1, \ldots, c_k \rangle$)
    $D$:= $\Sigma$
    output decode($c_1$)
    for $i$ := 2 to $k$ do
        if $c_i \in D$ then
            $D$:= $D \cup$ decode($c_{i-1}$) · decode($c_i$)[1]
        else
            $D$:= $D \cup$ decode($c_{i-1}$) · decode($c_{i-1}$)[1]
        output decode($c_i$)
```

Figure 9.19: LZ-decompression.

Chapter 10

Geometry

Geometrical algorithms cover a wide range of algorithms defined for sets of geometrical objects. These objects may include points, line segments, planes, curves and many other. In this chapter we will discuss a small selection of these elements, present fitting data structures and some algorithms that perform on these elements.

10.1 Application Domains

Geometric algorithms influence a lot of fields. The most prominent include computer graphics, robotics, geographic information systems (GIS) or computer aided design (CAD). Geometric algorithms enable us, to handle such amounts of data as is necessary for modern GIS or in the latest PC-game. Whether we have to perform hidden object removal in complex 3D-scenes or efficiently retrieve data about gas pipes within a given region, efficient data structures and algorithms are necessary. Typical questions for geometrical algorithms may include intersections between line segments, to calculate the convex hull of a set of points or localise a point in an partition of the d-dimensional space.
In this chapter we will study three distinct problems in detail. Those are line segment intersection, convex hull and the smallest enclosing circle problem.

10.2 Basic Elements and Definitions

Our algorithms will focus on two very basic elements. The elements we use are the points and line segments. For completeness we give following definitions:

Definition 17
*A **point** p is defined as a vector* $\in \mathbb{R}^d$*. We use* $p.i$ *to refer to the i-th element of the*

vector. In two- as well as three-dimensional space, we also use $p.x, p.y$ and $p.z$ to refer to the elements coordinates.

Definition 18
Given two points a and b, we define a ***line segment*** *$\overline{ab}$ as $\{\alpha \cdot a + (1 - \alpha) \cdot b | \alpha \in [0, 1]\}$.*

Definition 19
A ***polygon*** *in an d-dimensional space is a set of line segments given as a list of points $P = p_1, \ldots, p_n$ with $p_n = p_1$ and $\overline{p_i, p_{i+1}}$ for $i \in [1, n - 1]$ defining the outline of the polygon.*

We call a polygon ***convex*** *if for any two points a and b in P the line segment $\overline{ab}$ is contained in P.*

10.3 Data Structures

We will not study data structures in this place. We will only give a small example for geometric data structures as presented in Figure 10.1. The Figure shows the combination of the Voronoi diagram (dashed lines) and the Delaunay triangulation for a given set of points. Voronoi diagrams define the regions that are closest to one of the given points and therefore may be interesting whenever the trading area of a supermarket chain or something similar has to be computed. Triangulations become interesting whenever a plane has to be represented, as graphic cards need triangles to draw surfaces. The Delaunay triangulation of a set of points is a triangulation directly connected to the Voronoi diagram. The triangulation introduces one edge for any two vertices that share a common boundary in the Voronoi diagram. A Delaunay triangulation yields a “natural looking” triangulation which can be utilized for terrain rendering purposes for example.

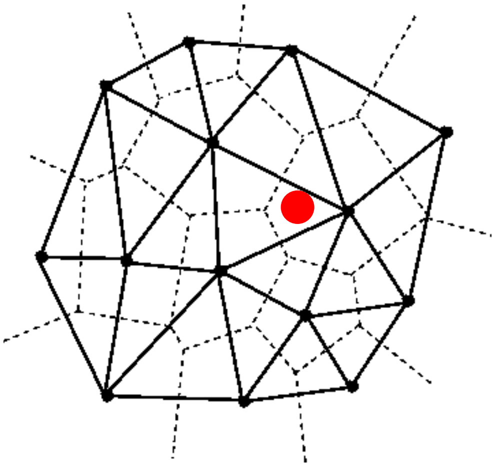


Figure 10.1: Voronoi Diagram and Delaunay Triangulation

10.4 Line Segment Intersection - Case Study

The line segment intersection problem can be defined as follows: Given a set $S = \{s_1, \ldots, s_n\}$ of line segments, calculate the set of all intersections $\bigcup_{s,t \in S} s \cap t$. First we will discuss the *orthogonal line segment intersection problem*. In this problem the line segments are layed out on an orthogonal grid.

10.4.1 Orthogonal Line Segment Intersection

This problem has its application domains in circuit development (where do conductors intersect) or in GIS (road crossings, bridges).

First we have a look at a naive implementation of the line segment intersection algorithm.

```
foreach $\{s, t\} \subseteq S$ do
    if $s \cap t \neq \emptyset$ then
        output $\{s, t\}$
```

The algorithm obviously performs in $\Theta(n^2)$ which is far to slow for large data sets. Naturally the questions arises: what is the best we can do? Since we have to look at each edge at least once and have to output every intersection, we get a minimal complexity of $\Omega(n + k)$ with k denoting the number of existing intersections. Without proof we give the lower bound for comparison based line segment intersection as $\Omega(n \cdot \log(n) + k)$. Obviously we can construct cases containing $\Theta(n^2)$ (see Figure 10.2) intersections, thus resulting in a running time of $\Theta(n^2)$ for any algorithm. Still most realistic inputs have $k = \mathrm{O}(n)$.

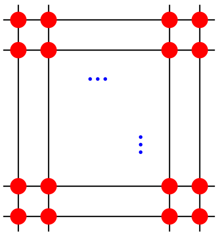


Figure 10.2: Layout with $\Theta(n^2)$ intersections

To improve our algorithm, we can take into account the spacial locality. Obviously, pairs of line segments, whose y-coordinates are far apart, cannot have a common point. Imagine all segments to be projected onto the y-axis: we only need to test line segments for intersection, whose y-intervals interlap. To find those pairs of lines, we imagine a horizontal line ℓ sweeping downwards over the plane. This line is called *sweep-line* and algorithms using this principle we call *plane-sweep* algorithms. While we sweep down the plane, we keep track of all vertical segments intersecting ℓ. This yields the following algorithm:

```
$T$=$\langle\rangle$ : SortedSequence of Segment
invariant $T$ stores the vertical segments intersecting $\ell$
$Q$:= sort($\langle(y,s)$ : ∃hor. segment $s$ at $y$ or ∃vert. segment $s$ starting/ending at $y\rangle$)
    //tie breaking: vert. starting events first, vert. finishing events last
foreach $(y,s) \in Q$ in descending order do
    if        $s$ is a vertical segment and starts at $y$ then $T$.insert($s$)
    else if   $s$ is a vertical segment and ends at $y$ then $T$.remove($s$)
    else      //we have a horizontal segment $s = \overline{(x_1,y)(x_2,y)}$
        foreach $t = \overline{(x,y_1)(x,y_2)} \in T$ with $x \in [x_1,x_2]$ do
            output $\{s,t\}$
        handle horizontal segments on $\ell$          // interval intersection problem
```

Keeping a sorted range T, the at most n insert and remove operations can be performed within $\mathrm{O}(\log(n))$ each. Finding the boundaries of the interval containing the intersections can be done in $\mathrm{O}(\log(n))$, the output of the intersections in $\mathrm{O}(k_s)$ with k_s denoting the number of intersections on the horizontal segment s. Overall this yields a running time of $\mathrm{O}(n \cdot \log(n) + \sum_s k_s) = \mathrm{O}(n \cdot \log(n) + k)$.

10.4.2 A more Complex Case

Now we will focus on a more complex case. We will now allow a more arbitrary positioning of the line segments while excluding some special cases. For now we do not allow horizontal line segments, overlapping segments and we do not allow one intersection to be a common intersection of more than two line segments.

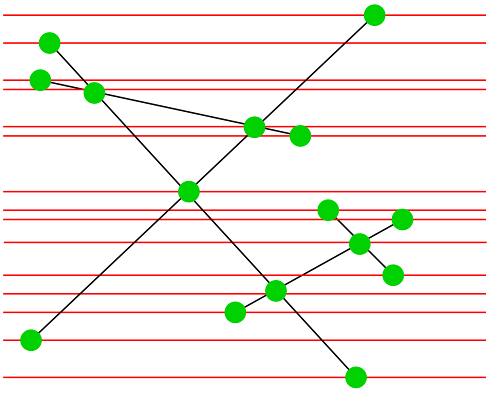


Figure 10.3: Events during the execution of the sweep-line algorithm

To extend our basic sweep-line algorithm for orthogonal line segment intersection, we introduce so called *events*. These events are illustrated in Figure 10.3. These events include all changes that affect our sweep-line algorithm in any way, to be exact they consist of segment starting points, segment end points and intersections. Using this definition, we only have to test for intersection of elements that are next to each other in T at some event.

Lemma 38
Let s_i and s_j be two non horizontal segments whose interiors intersect in a singe point p, and assume there is no third segment passing through p. Then there is an event point above p where s_i and s_j become adjacent and are tested for intersection.

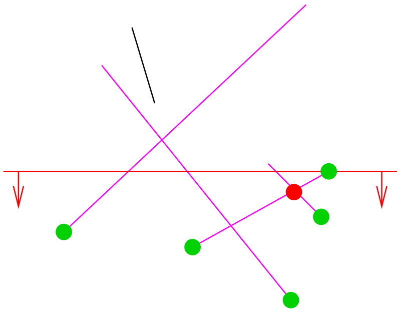
Proof.
Choose ℓ in such a way that there is no event located on ℓ and no event between ℓ and the horizontal line through p. Then s_i and s_j are adjacent in T for this position of ℓ. Since $T = \langle\rangle$ at the beginning of the algorithm, there exists an event where s_i and s_j become adjacent in T and are tested for intersection. □

This observations yields an improved version of our algorithm. We define two functions to aid our algorithm description. The first is called *findNewEvent* and calculates whether there is an intersection event between two given line segments.

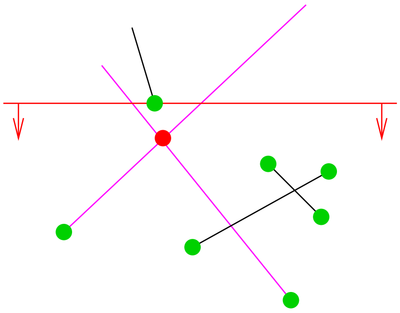
```
Function findNewEvent($s, t$)                                // $O(1 + \log n)$
    if $*s$ and $*t$ intersect at $y' < y$ then
        $Q$.insert(($y'$, intersection, $(s, t)$))
```

Due to the insert into Q, *findNewEvent* runs in $O(1 + log(n))$. The second function performs the event processing step. We distinguish between three version of *handleEvent* for the three different types of events.

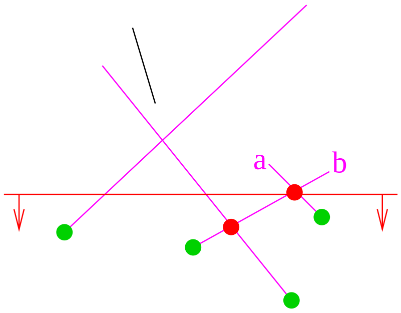
```
Function handleEvent($y$, start, $s, T, Q$)    // $n\times$
    $h$:= $T$.insert($s$)                      // $O(\log n)$
    prev:= pred($h$)                           // $O(1)$
    next:= succ($h$)                           // $O(1)$
    findNewEvent(prev, $h$)
    findNewEvent($h$, next)
```



```
Function handleEvent($y$, finish, $s, T, Q$)   // $n\times$
    $h$:= $T$.locate($s$)                      // $O(\log n)$
    prev:= pred($h$)                           // $O(1)$
    next:= succ($h$)                           // $O(1)$
    $T$.remove($s$)                            // $O(\log n)$
    findNewEvent(prev, next)
```



```
Function handleEvent($y$, intersection, $(a,b), T, Q$)// $k\times$
    output $(*s \cap *t)$                 // O(1)
    $T$.swap$(a,b)$                        // O(1)
    prev:= pred$(b)$                       // O(1)
    next:= succ$(a)$                       // O(1)
    findNewEvent(prev, $b$)
    findNewEvent($a$, next)
```



Due to the n line segments, we have to handle n start and n finish events. The number of intersection events is linear in the number of intersections as can be proven by an interpretation as a planar graph and by using eulers formula. For a detailed analysis we refer to [51].

```
$T$=$\langle\rangle$ : SortedSequence of Segment
invariant $T$ stores the relative order of the seqments intersecting $\ell$
$Q$ : MaxPriorityQueue
$Q$:= $Q \cup \left\{(y, \text{start}, s) : s = \overline{(x,y)(x',y')} \in S, y \le y'\right\}$      // O($n \log n$)
$Q$:= $Q \cup \left\{(y, \text{finish}, s) : s = \overline{(x,y)(x',y')} \in S, y \ge y'\right\}$     // O($n \log n$)
while $Q \neq \emptyset$ do
    $(y, \text{type}, s)$:= $Q$.deleteMin                            // O($(n+k)\log n$)
    handleEvent$(y, \text{type}, s, T, Q)$
```

The initialization of our algorithm performs in $\mathrm{O}(n \cdot \log(n))$, the event loop in $\mathrm{O}((n+k) \cdot \log(n))$. Thus the overall running time is in $\mathrm{O}((n+k) \cdot \log(n))$.

10.4.3 The Arbitrary Case

To allow horizontal lines, we have to introduce following order on Q:

$$(x,y) \prec (x',y') \Leftrightarrow y > y' \vee y = y' \wedge x < x'$$

This can be interpreted as a sweep-line with an infinitely small slope. With this we can once more adapt our algorithm:

To finally also allow overlapping segments, we can compute the line $g(s)$ for any segment $s \in S$ that contains s. Sorting S along $g(s)$ enables us to solve a 1-dimensional overlapping problem for each segment.

```
Function handleEvent($p = (x, y)$)
    $U$:= segments starting at $p$                              // from $Q$
    $C$:= segments with $p$ in their interior                   // from $T$
    $L$:= segments finishing at $p$                             // from $Q$
    if $|U| + |C| + |L| \geq 2$ then report intersection @ $p$
    $T$.remove($L \cup C$)
    $T$.insert($C \cup U$) such that order just below $p$ is correct
    if $U \cup C = \emptyset$ then
        findNewEvent($T$.findPred($p$), $T$.findSucc($p$), $p$)
    else
        findNewEvent($T$.findPred($p$), $T$.findLeftmost($p$), $p$)
        findNewEvent($T$.findRightmost($p$), $T$.findSucc($p$), $p$)
```

```
Function findNewEvent($s, t, p$)
    if $s$ and $t$ intersect at a point $p' \succ p$ then
        if $p' \notin Q$ then $Q$.insert($p'$)
```

10.4.4 Further Improvements

The algorithm as specified has one problem. The space consumption can become quite large, as any intersection is kept in the queue, even though the segments may not be adjacent at the moment. This yields a space consumption of $\Theta(n+k)$ which can be quite large for a large k. But we can improve this to $\mathrm{O}(n)$ by a simple modification. If we delete the intersection events from the queue for segments not longer adjacent, the space consumption obviously falls to $\mathrm{O}(n)$. The algorithm also does still report all intersections correctly, as all segments become adjacent again just before they intersect.

10.5 Convex Hull

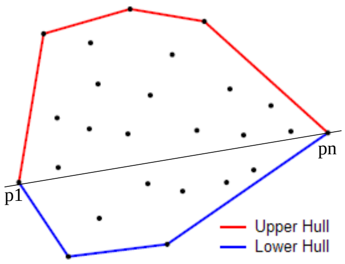


Figure 10.4: Upper/Lower Hull

In this section we will discuss the problem of the convex hull. For simplicity we focus on the 2-dimensional space $\mathbb{R}^2$.

The formal definition of the Convex-Hull-Problem in $\mathbb{R}^2$ is given as follows:

Given a set of points $P = p_1, \ldots, p_n \in \mathbb{R}^2$. Calculate a convex polygon C, using only points in P with $\forall p \in P : p$ contained in C. We discuss an algorithm running in $\mathrm{O}(sort(n))$ time. First we assume the points are given in lexicographical order (i.e. a point p_i comes before p_j if $p_i.x < p_j.x$ or

$p_i.x = p_j.x$ and $p_i.y < p_j.y$. If the points are not given in this order, we can sort them in $\mathrm{O}(sort(n))$ additional time.

For this algorithm, we first split the set of points along the line segment $\overline{p_1p_n}$. Now the algorithm can be implemented as two convex hull calculations that process the *lower-hull*, which represents the hull of points below $\overline{p_1p_n}$, and the *upper-hull* separately (illustrated in Figure 10.4). In the following we will focus on the upper-hull. The algorithm for the lower-hull can be formulated equivalently.

10.5.1 Graham Scan

Before we can actually discuss the algorithm, we have to give two definitions.

Definition 20

For a set of consecutive line segments, given as a set of points $P = p_1, \ldots, p_n$, we speak of a right turn at i if p_{i+1} is to the right of the line formed by p_{i-1} and p_i. In the same matter we speak of a left turn, if p_{i+1} is to the left of the line formed by p_{i-1} and p_i.

Looking only at the upper-hull, we observe that the convex-hull consists of a set of line segments performing only right-turns from p_1 to p_n and the line segment $\overline{p_1p_n}$. Thus, we can give the following algorithm to compute the upper-hull:

```
Function upperHull($p_1, \ldots, p_n$)
    $\mathcal{L}$=$\langle p_n, p_1, p_2\rangle$ : Stack of Point
    invariant $\mathcal{L}$ is the upper hull of $\langle p_n, p_1, \ldots, p_i\rangle$
    for $i := 3$ to $n$ do
        while ¬rightTurn($\mathcal{L}$.secondButlast, $\mathcal{L}$.last, $p_i$) do
            $\mathcal{L}$.pop
        $\mathcal{L}$:= $\mathcal{L} \circ \langle p_i\rangle$
    return $\mathcal{L}$
```

To prove our algorithm correct, we will prove the invariant.

Proof.

Before the *for-loop*, the set $\mathcal{L}$ contains only three vertices. Thus, $\mathcal{L}$ is a convex hull by definition. Now suppose that $\mathcal{L}$ forms the convex-hull of $\langle p_1, \ldots, p_{i-1}\rangle$ and consider the addition of p_i. After the execution of the *while-loop*, the points in $\mathcal{L}$ form a chain of right turns. To verify our assumption, we have to show that all points $\in \{p_1, \ldots, p_{i-1}\}$ not contained in $\mathcal{L}$ are below this chain. Due to the induction hypothesis, we know that a point below the new chain could only exist in the vertical slab between p_i and $p_i - 1$ since the new chain lies above the old chain. Because of the lexicographical order, no such point exists. □

The running time of the algorithm upper-hull itself lies in $\mathrm{O}(n)$. This is, as the *for* loop obviously runs in $\mathrm{O}(n)$ time. The additional work done by the *while* loop only adds an additional $\mathrm{O}(n)$ in total, as each point can only be deleted once from $\mathcal{L}$. Thus the overall running time is dominated by the sorting and the algorithm performs in $\mathrm{O}(n \cdot \log(n))$.
In three-dimensional space, the number of vertices does not directly imply a linear complexity of the convex hull. Still one can show that the number of edges and the number of faces are both in $\mathrm{O}(n)$. We refer to [51] for a proof. For even higher dimensions, the output complexity rises to $\mathrm{O}\left(n^{\lfloor d/2 \rfloor}\right)$.

10.6 Smallest Enclosing Circle

In this part we will go on to discuss the smallest enclosing circle problem which is defined as follows:

Given a set of points $P = \{p_1, \ldots, p_n\} \in \mathbb{R}^d$, calculate a sphere K with minimal radius so that $P \subseteq K$. We will discuss a simple algorithm with expected running time $\mathrm{O}(n)$ as introduced by Welzl in 1991. An illustration of the problem can be found in Figure 10.5.

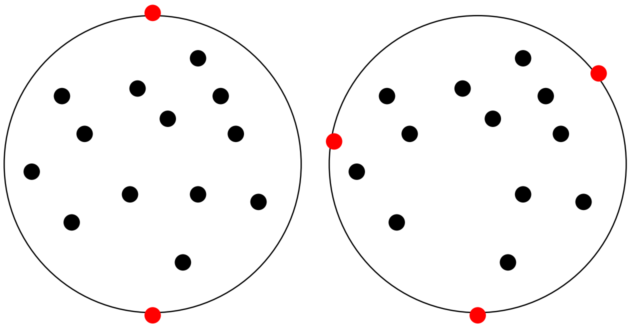


Figure 10.5: Enclosing Balls in 2-dimensional space

The algorithm is defined by a recursive function:

```
Function smallestEnclosingBallWithPoints($P, Q$)
    if $|P| = 1 \vee |Q| = d + 1$ then return ball($Q$)
    pick random $x \in P$
    $B$:= smallestEnclosingBallWithPoints($P \setminus \{x\}, Q$)
    if $x \in B$ then return $B$
    return smallestEnclosingBallWithPoints($P \setminus \{x\}, Q \cup \{x\}$)
```

Note the base of our recursion: a sphere for only one point can be given as the point itself without any radius. Also a set of $d+1$ vertices uniquely defines a sphere in d-dimensional space. In Q we store a set of border nodes that form the current sEB.

To prove our algorithm correct we have to prove two claims.

The first states:

Lemma 39
Smallest Enclosing Ball With Points (sEB) are non-ambiguous.

Proof.
Assume: $\exists$ sEB's $B_1 \neq B_2$. Since B_1 and B_2 are both sEB's $P \subseteq B_1 \wedge P \subseteq B_2$ holds. This directly induces $P \subseteq B_1 \cap B_2 \subseteq$ sEB($B_1 \cap B_2$) $= B$ and radius(B) $<$ radius(B_1). This cannot hold, since B_1 is a sEB. □

Lemma 40
If a point x is not in the sEB of $P\setminus\{x\}$, Q, then x is on the border of sEB(P).

Proof.
We show the contraposition: Assume x is not on the border of sEB(P). Then sEB(P) = sEB($P\setminus\{x\}$) = B, due to the non-ambiguous sEBs. □

10.6.1 Analysis

We have given a probabilistic algorithm for the smallest enclosing ball problem and proven it correct. In this section we want to analyse the expected running time for the algorithm.

We start by defining the base of our recurrence. As already mentioned above, our recursion $T(p,q)$ ends for $T(p,d+1) = T(1,p) = 0$.

For $T(p,q)$ following equation holds:
$T(p,q) \leq 1 + T(p-1,q) + \mathbb{P}[x \notin B] \cdot T(p,q+1)$ as we only have to execute the second recursive call, whenever $x \notin B$. Now let us inspect $\mathbb{P}[x \notin B]$ for a fixed subset of points $p_1,\ldots,p_i$ and its sEB Q. If we remove a point p_j, when does Q change? This happens only if we remove one of the $d+1$ points that uniquely define Q. If there are more than $d+1$ points on the boundary of Q, the smallest enclosing ball cannot change after the removal of one point. Thus, since we already know q points to be on the boundary of Q, the number of points that could change the sEB is given by $d+1-q$. Therefore $T(p,q) \leq 1 + T(p-1,q) + \frac{d+1-q}{p} \cdot T(p,q+1)$ holds.

Using this equation, we can calculate $T(p,d+1)$ for different values of d. For example $d=1$ would yield an expected running time of $3 \cdot n$ whereas $d=2$ would result in a running time of $10 \cdot n$. In general $T(p,0) \leq d! \cdot n$ holds.

10.7 Range Search

The most general version of range searching problem is given as follows: Given a set of geometric objects P and a geometric range Q, determine which objects of P Q contains. In short we will write $P \cap Q$ for such a *range reporting* query. Furthermore we might be interested only in the number of elements contained in Q. We call this a *range counting* query and write $|P \cap Q|$. In most applications of range search, we get a lot of querys for a given set of points. Thus we exploit precomputation to answer the queries as fast as possible. Intuitively we would expect the precomputation the be possible in $\mathrm{O}(n \log n)$ and consume only $\mathrm{O}(n)$ space. Counting queries should be possible to answer in $\mathrm{O}(\log n)$ while we would expect range reporting queries to be possible in $\mathrm{O}(k + \log n)$ or at least in $\mathrm{O}(k \log n)$ with k denoting the number of contained elements.

10.7.1 1D Range Search

In the 1-dimensional case, we consider a list of numbers. To answer range reporting and range counting queries, we utilize a binary range tree, as depicted in Figure 10.6. The tree consists of a doubly linked list containing the given elements in sorted order. On top we keep a tree structure, containing the splitting element for the underlying range as key.

The construction is obviously bounded by sorting the elements in $\mathrm{O}(n \log n)$ time. Range queries traverse the tree for both ends of the range, giving the start and the end of the list, containing all elements contained in the range. This can be done in $\mathrm{O}(\log n)$. Afterwards, the range can then be reported in $\mathrm{O}(k)$. Range counting queries can be answered in $\mathrm{O}(\log n)$ if we keep the number of elements contained in the range covered by a given tree node up-to-date.

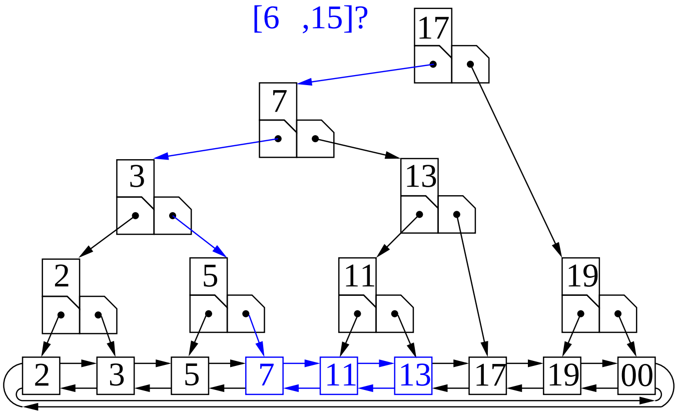


Figure 10.6: A 1D range tree.

10.7.2 2D Range Search

Range queries in 2-dimensional space have their application in geocoding or collision detection. A possible representation for answering those kind of queries fast is the Wavelet Tree introduced by Chazelle in 1988. The wavelet tree utilizes bitvectors to represent itself.

Be reminded that in the following, coordinates are represented by their ranks in each dimension respectively, i. e. we have only integers. On each level of the tree, these ranks are newly calculated, i. e. compacted. Futhermore, the sequence of points is always kept sorted in y-direction.

Definition 21
For a given range of elements $X = \langle x_1, \ldots, x_n \rangle$, a Wavelet Tree *is recursively defined as follows. The tree keeps a bitvector b with $b[i] = 1$ iff $x_i > \lfloor n/2 \rfloor$. Furthermore, it contains two children, each recursivly defined as the Wavelet Trees of $\langle x_i : x_i \leq \lfloor n/2 \rfloor \rangle$ as the left child and $\langle x_i - \lfloor n/2 \rfloor : x_i > \lfloor n/2 \rfloor \rangle$ as the right child respectively. This recursion is stopped as soon as $n \leq n_0$, where n_0 is a tuning parameter. In this case, the sequence is stored directly. A sample wavelet tree is shown in Figure 10.7.*

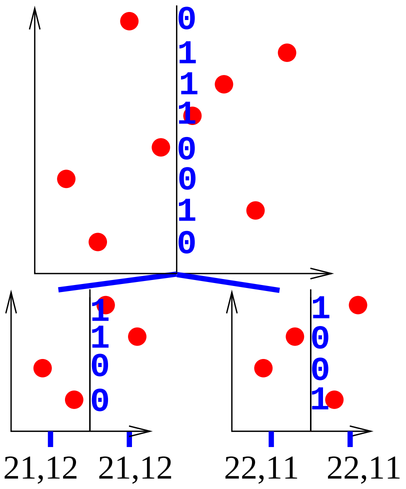


Figure 10.7: Example Wavelet Tree

Definition 22
*The Dominance Counting Problem is defined as follows: Given a sequence of elements $X = \langle x_1, \ldots, x_n \rangle$ as well as an pair (x, y),
find $|\{x_i : x_i \in X \wedge x_i \geq x \wedge i \geq y\}|$.* (dominance criterion)

Using Wavelet Trees, we can solve this problem with Algorithm 10.8. In every recursion step, the number of elements y_r less or equal to y is calculated using the rank function of the bitvector. How this can be accomplished will be discussed later. If the desired value of x is found in the left child, we know of at least $\lceil n/2 \rceil - y_r$ elements to fulfill the dominance criterion. The rest can be calculated recursively with $(x, y - y_r)$ as input for the left child, as all (x_i, y_i) with $y_i < y \wedge x_i > x$ are not considered in the left child. The recursion for $x > \lfloor n/2 \rfloor$ can be explained in a similar matter. For an illustration, the reader can turn to Figure 10.9.

If we implement the rank to work in constant time (see below), the algorithm runs in $\mathrm{O}(\log n)$ as we only follow one recursive path and have a recursion depth of $\mathrm{O}(\log n)$.

```
Function intDominanceCount($x, y$)                         // $|[x,n] \times [y,n] \cap P|$
    if $n \le n_0$ then return $|[x,n] \times [y,n] \cap P|$              // brute force
    $y_r$:= $b$.rank($y$)                       // Number of els $\le y$ in right half
    if $x \le \lfloor n/2 \rfloor$ then
        return $\ell$.intDominanceCount($x, y - y_r$) + $\lceil n/2 \rceil - y_r$
    else
        return $r$.intDominanceCount($x - \lfloor n/2 \rfloor, y_r$)
```

Figure 10.8: Algorithm intDominanceCount

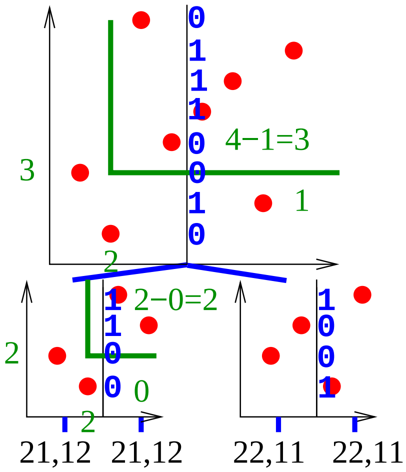


Figure 10.9: Example query for dominance counting using a Wavelet Tree

Using Algorithm 10.10, we can also perform range counting queries on integer values. The algorithm is illustrated in Figure 10.11.

```
Function intRangeCount($[x, x'] \times [y, y']$)
    return
        intDominanceCount$(x, y)-$
        intDominanceCount$(x', y)-$
        intDominanceCount$(x, y')+$
        intDominanceCount$(x', y')$
```

Figure 10.10: Algorithm: integer range counting

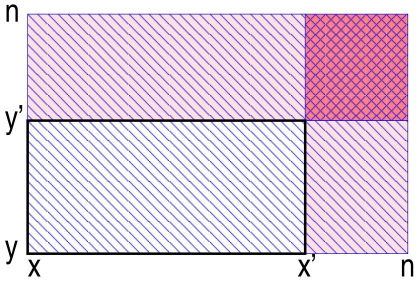


Figure 10.11: Integer Range Counting using Integer Dominance Counting

Wavelet Tree Dominance Reporting Query

To implement the Dominance Reporting Query, we look at a subproblem first.

Definition 23
The One Sided Reporting problem is given as: Given a set $P = \{(x_1, y_1), \ldots (x_n, y_n)\}$ and a value y, find $P \cap [1, n] \times [y, n]$.

The One Sided Reporting problem can be solved by Algorithm 10.12. It is based on the same principles as Algorithm 10.8. The only difference is that we have to consider all elements above the given y value. Thus we have to perform a recursion step in both children.

```
Function oneSidedReporting($y$)                          // $|[1, n] \times [y, n] \cap P|$
    if $n \leq n_0$ then return $|[1, n] \times [y, n] \cap P|$          // brute force
    $y_r$:= $b$.rank($y$)                       // Number of els $\leq y$ in right half
    $R$:= $\emptyset$
    if $y_r < \frac{n}{2}$ then $R$:= $R \cup r$.oneSidedReporting($y_r$)
    if $y - y_r < \frac{n}{2}$ then $R$:= $R \cup \ell$.oneSidedReporting($y - y_r$)
    return $R$
```

Figure 10.12: Algorithm for One Sided Reporting in a Wavelet Tree

Having solved the One Sided Reporting problem, we are ready to report all elements dominating an (x, y) pair with Algorithm 10.13:

```
Function intDominanceReporting$(x, y)$                                   // $|[x, n] \times [y, n] \cap P|$
    if $n \leq n_0$ then return $|[x, n] \times [y, n] \cap P|$                          // brute force
    $R$:= $\emptyset$                                                              // Result
    $y_r$:= $b$.rank$(y)$                                    // Number of els $\leq y$ in right half
    if $x \leq \lfloor n/2 \rfloor$ then                                        // Both halves interesting
        if $y - y_r < \frac{n}{2}$ then $R$:= $R \cup \ell$.intDominanceReporting$(x, y - y_r)$
        if $y_r < \frac{n}{2}$ then $R$:= $R \cup r$.oneSidedReporting$(y_r)$
    else if $y_r < \frac{n}{2}$ then $R$:= $R \cup r$.intDominanceReporting$(x - \lfloor n/2 \rfloor, y_r)$
    return $R$
```

Figure 10.13: Algorithm for Dominance Reporting using a Wavelet Tree

A generalization to the range reporting query can be achieved. The range reporting query, as all other queries before, is defined recursively. We report elements in the moment that all elements above a given y value are dominant. The rest of the algorithm is identical to the range counting algorithm.

We can adapt the one sided reporting to work with a minimal and a maximal rank and report the elements within a range. The rest of the adaption is trivial following the Illustration in Figure 10.14.

2D Range Search on floating point numbers

Now consider the following problem:

Definition 24
The 2D-Range Search problem is defined as follows.
Given a set of data $P = \{p_1, \dots, p_n\} \subseteq \mathbb{R}^2$*, and an axis parallel rectangle* $Q = [x, x'] \times [y, y']$

1. *find* $P \cap Q$ (range query)

2. *find* $|P \cap Q|$ (range counting)

We can reduce the general problem onto the integer problem on $[1, n] \times [1, n]$. This can be achieved by replacing each pair $(x, y) \in P$ with $(\mathrm{rank(x, x_i)}, \mathrm{rank(y, y_i)})$, where $\mathrm{rank}(x, x_i)$ denotes the rank of x in a sorted range of the $\{x_i | (x_i, y) \in P \text{ for } y \in \mathbb{R}\}$. This replacement can be achieved by sorting the x_i and y_i respectivly im combination with a binary search for every x_i, y_i in the resulting ranges. Thus we can achieve a general range query in $\mathrm{O}(n \log n)$, using the integer range counting methods discussed above.

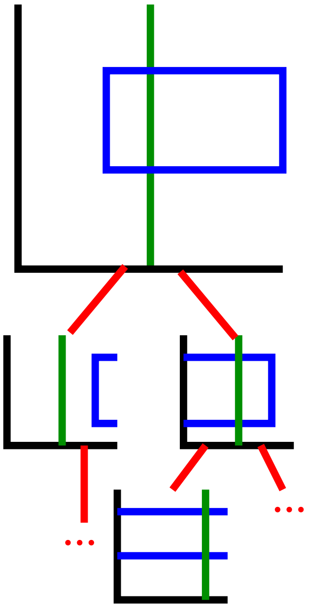


Figure 10.14: Illustration of general range reporting Query

10.7.3 Bitvectors

Bitvectors, as used in the range query algorithms discussed above, store a set of bits and offer a set of distinct operations. For our use case, the interesting operation is the $\mathrm{rank}(j)$ operation.

To support the rank operation in constant time, we precompute the ranks of every iB-th element, were B can be seen as a tuning parameter. Than $rank(j)$ can be computed as the sum of the precomputed value $j \mod B$ and $rank(v[j \mod B], j])$. If we chose $B = \Theta(\log n)$, we use $\mathrm{O}(n)$ time and $\mathrm{O}(n)$ bits. This process is illustrated in Figure 10.15.

The constant lookup can be either realized using a lookup table or by the instruction population count which on some systems (e. g. SSE 4.2) counts the number of set bits in a machine word.

In a similar manner, we can calculate the position of the i-th set bit $b.\mathrm{select}(i)$ in constant time. Furthermore, it is possible to reach the information theoretical optimal space of $n + o(n)$ bits.

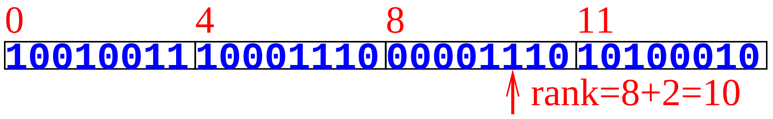


Figure 10.15: Illustration of the rank function on a bit vector

Chapter 11

Approximation Algorithms

Although NP-hard optimization problems do not offer footholds for finding optimal solutions efficiently, they may still offer footholds for finding near-optimal solutions efficiently. An NP-optimization problem Π is either a minimization or a maximization problem. Each valid instance I of Π comes with a nonempty set of feasible solutions, each of which is assigned a nonnegative rational number called its objective function value. There exist polynomial time algorithms for determining validity, feasibility, and the objective function value. A feasible solution that achieves the optimal objective function value is called an optimal solution. $f(\mathbf{x}^*(I))$ will denote the objective function value of an optimal solution $\mathbf{x}^*(I)$ to instance I. An approximation algorithm, $\mathcal{A}$, for Π produces, in polynomial time, a feasible solution $\mathbf{x}(I)$ whose objective function value is "close" to the optimal; by "close" we mean within a guaranteed factor of the optimal:

Definition 25
An approximation algorithm for a NP-minimization problem Π has approximation ratio ρ *if and only if it finds for every valid instance I a feasible solution $\mathbf{x}(I)$ within*

$$\frac{f(\mathbf{x}(I))}{f(\mathbf{x}^*(I))} \leq \rho \ .$$

11.1 Minimum Makespan Scheduling

A central problem in scheduling theory is the following.

Definition 26
Give processing times for n jobs, $t_1, t_2, \ldots, t_n$, and an integer m, the minimum makespan problem *is to find an assignment $x : \{1, \ldots, n\} \to \{1, \ldots, m\}$ of the jobs to m identical machines so that the completion time, also called the* makespan

$L_{\max}$ is minimized. Let

$$L_i := \sum_{x(j)=i} t_j$$

be the load *of machine i, then*

$$L_{\max} := \max_{i=1}^{m} L_i$$

We will give a simple factor 2 algorithm for this problem.

11.1.1 List Scheduling

The algorithm (Figure 11.1) is very simple: schedule the jobs one by one, in an arbitrary order, each job being assigned to a machine with least amount of work so far [35]. This algorithm is based on the following two lower bounds on the makespan, $L_{\max}$:

Lemma 41
The average time for which a machine has to run, $\left(\sum_{j=1}^{n} t_j\right)/m \leq L_{\max}$.

Lemma 42
The largest processing time, $\max_{j=1}^{n} t_j \leq L_{\max}$ and thus $\forall j \in \{1,..,n\} : t_j \leq L_{max}$.

```
ListScheduling($n$, $m$, $\mathbf{t}$)
    $J := \{1, \ldots, n\}$
    array $L[1..m] = [0, \ldots, 0]$
    while $J \neq \emptyset$ do
        pick any $j \in J$
        $J := J \setminus \{j\}$
        // Shortest Queue:
        pick $i$ such that $L[i]$ is minimized
        $\mathbf{x}(j) := i$
        $L[i] := L[i] + t_j$
    return $\mathbf{x}$
```

Figure 11.1: List Scheduling

Lemma 43
Let ℓ be the job that finished last on machine i, then

$$L_{\max} \leq \sum_{j=1}^{n} \frac{t_j}{m} + \frac{m-1}{m} t_\ell \ .$$

Proof.
Let i be the machine that job ℓ is assigned to. Let t be the load of machine i excluding job ℓ. As machine i has not more load than any other machine, it must hold

$$t \leq \sum_{j \neq \ell} \frac{t_j}{m}$$

and therefore, we can follow that

$$L_{\max} = t + t_\ell \leq \sum_{j \neq \ell} \frac{t_j}{m} + t_\ell = \sum_j \frac{t_j}{m} + \frac{m-1}{m} t_\ell$$

□

Theorem 44
ListScheduling has an approximation ratio of $2 - \frac{1}{m}$.

Proof.
Let $\mathbf{x}$ be the solution of ListScheduling and $\mathbf{x}^*$ an optimal solution. Then

$$\begin{aligned} \frac{f(\mathbf{x})}{f(\mathbf{x}^*)} &\overset{\text{Lemma 43}}{\leq} \frac{\sum_j t_j/m}{f(\mathbf{x}^*)} + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \\ &\overset{\text{Lemma 41}}{\leq} 1 + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \\ &\overset{\text{Lemma 42}}{\leq} 1 + \frac{m-1}{m} = 2 - \frac{1}{m} \end{aligned}$$

□

A tight example of this algorithm is provided by a sequence of m^2 jobs with unit processing time, followed by a single job of length m, see Figure 11.2. The schedule obtained by the algorithm has a makespan of $2m$, while the optimal makespan is $m + 1$.

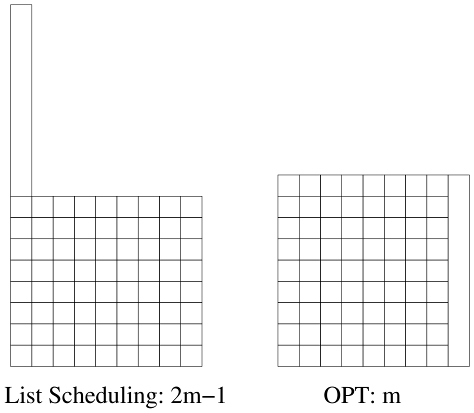


Figure 11.2: Worst-case example for ListScheduling.

11.1.2 More Scheduling

- 4/3-approximation: Sort jobs descending by processing time, then use ListScheduling [36].
- PTAS [38].
- 2-approximation for unrelated machines: Job j has processing time t_{ji} on machine i [50].

11.2 Traveling Salesman Problem

Definition 27
Give a complete graph $G = (V, V \times V)$ with nonnegative edge costs $d(u, v)$. The traveling salesman problem (TSP) *is to find a minimum cost cycle visiting every vertex exactly once.*

11.2.1 Nonapproximability of General TSP

It its full generality, TSP cannot be approximated, assuming P $\neq \mathrm{NP}$.

Theorem 45
TSP cannot be approximated within a factor of A, unless $\mathrm{P} = \mathrm{NP}$.

Proof.
Assume, for a contradiction, that there is a factor a polynomial time approximation algorithm for the general TSP problem. We will show that this algorithm can be used for deciding the Hamiltonian cycle problem (which is NP-hard) in polynomial time, thus implying $\mathrm{P} = \mathrm{NP}$.

The central idea is a reduction from the Hamiltonian cycle problem to the TSP, that transforms a graph G on n vertices to an edge-weighted complete graph G' on n vertices such that

- if G has a Hamiltonian cycle, then the cost of an optimal TSP tour in G' is n, and
- if G does not have a Hamiltonian cycle, then an optimal TSP tour in G' is of cost $> an$.

Observe that when we run on graph G', our approximation algorithm must return a solution of cost $\leq an$ in the first case, and a solution of cost $> an$ in the second case. Thus, it can be used for deciding whether G contains a Hamiltonian cycle.

The reduction is simple. Assign a weight of 1 to edges of G, and a weight of an to nonedges, to obtain G'. Now, if G has a Hamiltonian cycle, then the corresponding tour in G' has a cost n. On the other hand, if G has no Hamiltonian cycle, any tour in G' must use an edge of cost an, and therefore has cost $> an$. □

11.2.2 Metric TSP

Notice that in order to obtain the strong nonapproximability result of the previous section, we had to assign edge costs that violate triangle inequality, i.e., consider *metric* TSP, the problem remains NP-complete, but is no longer hard to approximate.

Definition 28
A undirected graph G fulfills the triangle inequality *if and only if*

$$\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$$

Let $G = (V, E)$ be an arbitrary undirected weighted graph. Replace for each edge the weight by the shortest path distance between its endpoints. This results in a graph G fulfilling the triangle inequality, called the *metric completion* of G.

We will present a simple factor 2 algorithm for the metric TSP (Figure 11.3). The lower bound we will use for obtaining this factor is the cost of an MST in G. This is a lower bound because deleting any edge from an optimal solution to TSP gives us a spanning tree of G.

```
MetricTSPApprox($G$)
    $T$:= MST($G$)                                   // weight($T$) $\leq$opt
    $T'$:= $T$ with every edge doubled                // weight($T'$) $\leq$2opt
    $T''$:= EulerTour($T'$)                           // weight($T''$) $\leq$ 2opt
    output the tour that visits vertices in $G$ in     // shortcutting
    the order of their first appearance in $T''$
```

Figure 11.3: 2-approximation of metric TSP

Definition 29
Let $G = (V, E)$ be a (multi)-graph with $|E| = m$. A path $P = \langle e_1, \ldots, e_m \rangle$ is an Eulerian tour *if and only if every edge is visited exactly once, that is $\{e_1, \ldots, e_m\} = E$.*

A graph G has a Eulerian tour if and only if the degree of every node in v is even. Euler proved this in 1736. Therefore, by doubling every edge, we know that the graph T' has a Eulerian tour.

Theorem 46
MetricTSPApprox is a factor 2 approximation algorithm for metric TSP.

Proof.
Let opt be the cost of an optimal tour. As noted above, weight$(T) \leq$ opt. Since T' contains each edge of T twice, weight$(T') \leq 2$opt. Because of triangle inequality, after the "shortcutting" step, the weight did not increase. □

Figure 11.4 shows an example of approximating TSP with MST.

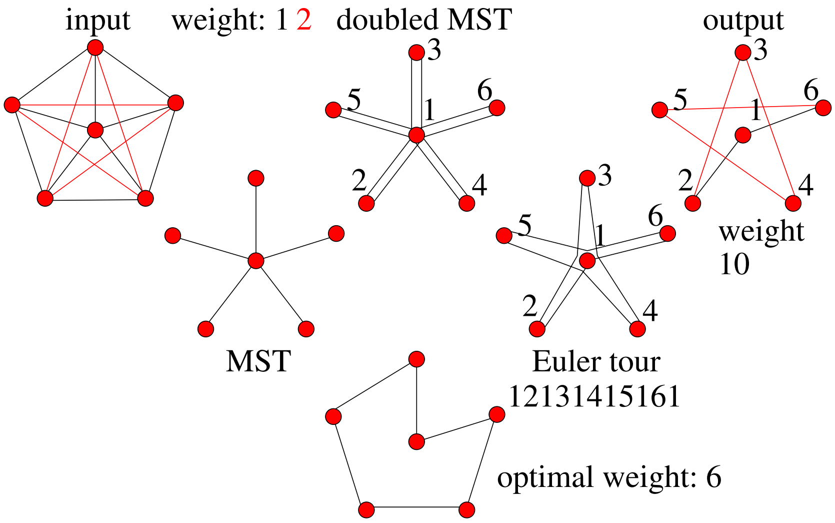


Figure 11.4: Example metric TSP approximation

11.2.3 Further Notes on TSPs

- A 3/2-approximation can be found in [19].

11.3 Knapsack

Definition 30
Give n objects, with specified sizes $w_i \in \mathbb{N}$ and profits $p_i \in \mathbb{N}$, and a "knapsack capacity" $W \in \mathbb{N}$. The knapsack problem *is to find a subset of objects whose total size is bounded by W and total profit is maximized.*

An obvious algorithm for this problem is to sort the objects by decreasing ratio of profit to size, and then greedily pick objects in this order. However, this algorithm can be made to perform arbitrarily badly.

11.3.1 Pseudo-Polynomial Dynamic Programming

For any optimization problem Π, an instance consists of *objects*, such as sets of graphs, and *numbers*, such as cost, profit, size, etc. So far, we have assumed that all numbers occurring in a problem instance I are written in binary. The *size* of the instance, denoted by $|I|$, was defined as the number of bits needed to write I under this assumption. An algorithm for problem Π is said to be efficient if its running time on instance I is bounded by a polynomial in $|I|$. Let us consider the following weaker definition.

Definition 31
Let n_u be the unary size *of instance I, defined as the number of bits needed to write I in unary. An algorithm for problem Π whose running time on instance I is bounded by a polynomial in n_u will be called a* pseudo-polynomial time algorithm.

The knapsack problem, being NP-hard, does not admit a polynomial time algorithm (unless $\mathrm{P} = \mathrm{NP}$); however, it does admit a pseudo-polynomial time algorithm. All known pseudo-polynomial time algorithms are based on dynamic programming.

Let $\hat{P}$ be an upper bound on the profit, e.g., $\sum_i p_i$ is a trivial upper bound that can be achieved by any solution. For each $i \in \{1, \ldots, n\}$ and $P \in \left\{1, \ldots, \hat{P}\right\}$, let $C(i, p)$ denote the smallest size of a subset of the objects $1, \ldots, i$ that have profit $\geq P$, and ∞ if no such set exists. Clearly, $C(i, P)$ is known for every $P \in \left\{1, \ldots, \hat{P}\right\}$. The following recurrence helps to compute all values $C(i, P)$ in $\mathrm{O}\big(n\hat{P}\big)$ time:

$$C(i, P) = \min\left(C(i-1, P), C(i-1, P-p_i) + w_i\right) \ .$$

The maximum profit achievable by objects of total size bounded by W is $\max\{P \mid C(n, P) \leq W\}$. We thus get a pseudo-polynomial time algorithm for knapsack. An efficient implementation does only need to store one $C(\cdot, P)$ at a time, thus requiring only $\hat{P} + \mathrm{O}(n)$ machine words space. To reconstruct the solution for each $C(i, P)$, $n\hat{P}$ decision bits are additionally necessary, storing whether $C(i-1, P)$ or $C(i-1, P-p_i) + w_i$ composed the minimum in the above recurrence.

11.3.2 Fully Polynomial Time Approximation Scheme

Some NP-hard optimization problems, including knapsack, allow approximability to any required degree. Let Π be an NP-hard optimization problem with objective function f. We will say that algorithm $\mathcal{A}$ is an *approximation scheme* for Π if on input (I, ε), where I is an instance of Π and $\varepsilon > 0$ is an error parameter, it outputs a solution $\mathbf{x}$ such that:

- $f(\mathbf{x}) \leq (1+\varepsilon)\cdot$opt if Π is a minimization problem.
- $f(\mathbf{x}) \geq (1-\varepsilon)\cdot$opt if Π is a maximization problem.

$\mathcal{A}$ will be said to be a *polynomial time approximation scheme*, abbreviated PTAS, if for each fixed $\varepsilon > 0$, its running time is bounded by a polynomial in $|I|$.

The definition given above allows the running time of $\mathcal{A}$ to depend arbitrarily on ε. This is rectified in the following more stringent notion of approximability. If the previous definition is modified to require that the running time of $\mathcal{A}$ be bounded by a polynomial in $|I|$ and $1/\varepsilon$, then $\mathcal{A}$ will be said to be a *fully polynomial time approximation scheme*, abbreviated FPTAS.

In a very technical sense, an FPTAS is the best one can hope for an NP-hard optimization problem, assuming $\mathrm{P} \neq \mathrm{NP}$. The knapsack problem admits an FPTAS [41].

Notice that if the profits of objects were small numbers, i.e., they were bounded by a polynomial n, then the algorithm described in Section 11.3.1 would be a regular polynomial time algorithm, since its running time would be bounded by a polynomial in $|I|$. The key idea behind obtaining an FPTAS is to exploit precisely this fact: we will ignore a certain number of least significant bits of profits of objects (depending on the error parameter ε), so that the modified profits can be viewed as numbers bounded by a polynomial in n and $1/\varepsilon$. This will enable us to find a solution whose profit is at least $(1-\varepsilon)\cdot$opt in time bounded by a polynomial in n and $1/\varepsilon$.

```
KnapsackFPTAS($\mathbf{w} = \langle w_1, \ldots, w_n \rangle$, $\mathbf{p} = \langle p_1, \ldots, p_n \rangle$, $W$)
    $P$:= $\max_i p_i$                                          // maximaler Profit
    $K$:= $\frac{\varepsilon P}{n}$                              // Skalierungsfaktor
    $\mathbf{p}' := \langle p'_1, \ldots, p'_n \rangle$ with $p'_i$:= $\left\lfloor \frac{p_i}{K} \right\rfloor$       // skaliere Profite
    $\mathbf{x}'$:= dynamicProgrammingByProfit($\mathbf{p}', \mathbf{w}, C$)
    output $\mathbf{x}'$
```

Figure 11.5: FPTAS for Knapsack

We see the output $\mathbf{x}'$ both as set of objects i comprising the solution, and 0/1 bit-vector with an 1 at the i^{th} position indicating that object i is in the solution.

Lemma 47
$\mathbf{p} \cdot \mathbf{x}' \geq (1-\varepsilon)\cdot$*opt*.

Proof.
Let $\mathbf{x}^*$ denote the optimal solution. For any object i, because of rounding down,

Kp'_i can be smaller than p_i, but by not more than K. Therefore,

$$\begin{array}{rcl}\mathbf{p}\cdot\mathbf{x}^* - K\mathbf{p}'\cdot\mathbf{x}^* & = & \sum_{i\in\mathbf{x}^*}\left(p_i - K\left\lfloor\frac{p_i}{K}\right\rfloor\right) \\ & \leq & \sum_{i\in\mathbf{x}^*}\left(p_i - K\left(\frac{p_i}{K}-1\right)\right) = |\mathbf{x}^*|K \leq nK\end{array}$$

And thus $K\mathbf{p}'\cdot\mathbf{x}^* \geq \mathbf{p}\cdot\mathbf{x}^* - nK$. The dynamic programming step must return a solution $\mathbf{x}'$ at least as good as $\mathbf{x}$ under the new profits. Therefore,

$$K\mathbf{p}'\cdot\mathbf{x}^* \leq K\mathbf{p}'\cdot\mathbf{x}' = \sum_{i\in\mathbf{x}'} K\left\lfloor\frac{p_i}{K}\right\rfloor \leq \sum_{i\in\mathbf{x}'} K\frac{p_i}{K} = \mathbf{p}\cdot\mathbf{x}'$$

Combining both inequalities yields

$$\mathbf{p}\cdot\mathbf{x}' \geq K\mathbf{p}'\cdot\mathbf{x}^* \geq \mathbf{p}\cdot\mathbf{x}^* - nK = \mathrm{opt} - \varepsilon P \geq (1-\varepsilon)\mathrm{opt}$$

□

11.3.3 More Knapsack

The best currently known FPTAS [47] has a running time of

$$\mathrm{O}\left(\min\left\{n\log\frac{1}{\varepsilon} + \frac{\log^2\frac{1}{\varepsilon}}{\varepsilon^3}, \ldots\right\}\right)$$

11.4 Book References

This chapter is largely based on Vazirani [85]. The book by Wanka [88] introduces into algorithms and is written in German. However, not all of the content of the course are covered. A third book is by Hromkovic [40].

Chapter 12

Fixed-Parameter-Algorithmen

In diesem Kapitel betrachten wir die folgenden drei Probleme:

INDEPENDENT SET

Gegeben: Graph $G = (V, E)$, $k \in \mathbb{N}$

Gesucht: Unabhängige Menge $V' \subseteq V$ mit $|V'| \geq k$

Eine Menge $V' \subseteq V$ heißt *unabhängig* genau dann, wenn für alle $v, w \in V'$ gilt $\{v, w\} \notin E$ (siehe Abbildung 12.1).

VERTEX COVER

Gegeben: Graph $G = (V, E)$, $k \in \mathbb{N}$

Gesucht: Vertex Cover $V' \subseteq V$ mit $|V'| \leq k$

Eine Menge $V' \subseteq V$ heißt *Vertex Cover* genau dann, wenn für jede Kante $\{v, w\} \in E$ gilt $u \in V'$ oder $v \in V'$ (siehe Abbildung 12.1).

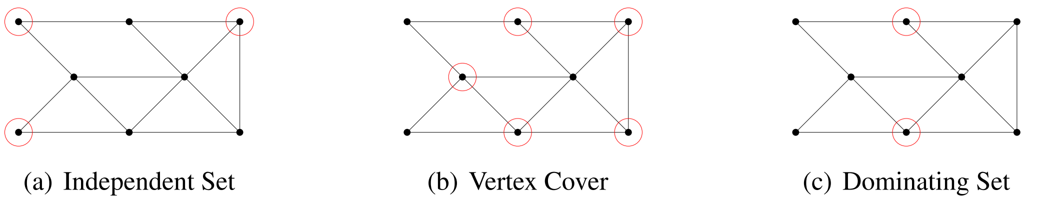
DOMINATING SET

Gegeben: Graph $G = (V, E)$, $k \in \mathbb{N}$

Gesucht: Dominating Set $V' \subseteq V$ mit $|V'| \leq k$

Eine Menge $V' \subseteq V$ heißt *Dominating Set* genau dann, wenn für jeden Knoten $v \in V$ gilt, dass entweder v selbst oder einer seiner Nachbarn in V' enthalten ist (siehe Abbildung 12.1).

Diese Probleme sind $\mathbf{NP}$-schwer und lassen sich durch erschöpfendes Aufzählen (Brute-Force-Methode) aller $\binom{n}{k}$ Teilmengen $V' \subseteq V$ mit $|V'| = k$ mit einer Laufzeit in $\mathrm{O}\big(n^k \cdot (n + m)\big)$ entscheiden.



(a) Independent Set (b) Vertex Cover (c) Dominating Set

Figure 12.1: Independent Set, Vertex Cover und Domintating Set eines Beispiel-Graphen.

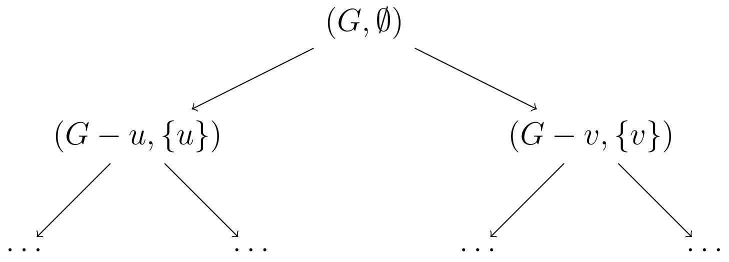
Im folgenden wird ein alternativer Algorithmus für Vertex-Cover vorgestellt. Es handelt sich dabei um einen parametrisierten Algorithmus, der auf einem tiefenbeschränkten Suchbaum basiert. Die Vorgehensweise des Algorithmus ist wie folgt:

1. Konstruiere vollständigen binären Baum der Tiefe k.

2. Markiere die Wurzel des Baumes mit $(G, \emptyset)$.

3. Jeder Knoten wird wie folgt rekursiv markiert: Sei v ein mit (H, S) markierter Knoten des Baumes, der zwei nicht markierte Kinder hat. Wähle eine beliebige Kante $\{u, v\}$ aus der Kantenmenge von H und markiere das linke Kind mit $(H - u, S \cup \{u\})$ sowie das rechte Kind mit $(H - v, S \cup \{v\})$. Dabei bezeichne $H - v$ den Graphen, der durch Löschen des Knotens v und aller inzidenten Kanten aus H entsteht.

4. Falls es einen Baumknoten mit der Markierung (H, S') existiert, wobei H der leere Graph ist, dann ist S' ein Vertex Cover der Größe $|V'| \leq k$. Andernfalls existiert kein Vertex Cover mit k oder weniger Knoten für G.



Die Laufzeit bei diesem Ansatz ist in $\mathrm{O}\big(2^k \cdot (n + m)\big)$ und damit von der Form $\mathrm{O}(f(k) \cdot p(n))$, wobei p ein Polynom und $f : \mathbb{N} \to \mathbb{N}$ eine berechenbare Funktion ist, die nur von k abhängt.

Ein Optimierungsproblem Π kann in kanonischer Weise als *parametrisiertes Problem* aufgefasst werden: Zu einer Instanz $\mathcal{I}$ von Π und einem Parameter $k \in \mathbb{N}$

wird eine zulässige Lösung gesucht, deren Wert mindestens (Maximierungsproblem) bzw. höchstens (Minimierungsproblem) k ist.

Definition 32 (fixed parameter tractable)
Ein parametrisiertes Problem Π *heißt* fixed parameter tractable, *wenn es einen Lösungsalgorithmus zu* Π *mit Laufzeit* $\mathrm{O}(f(k) \cdot p(n))$ *gibt. Dabei ist* n *die Eingabegröße,* p *ein Polynom,* k *der Parameter und* f *eine berechenbare Funktion, die nur von* k *abhängt.*

Definition 33 (FPT)
Die Klasse FPT ist die Klasse aller Probleme, die fixed parameter tractable sind.

Remark 48
VERTEX COVER *ist in FPT. Für* INDEPENDENT SET *und* DOMINATING SET *ist nicht bekannt, ob sie in FPT sind. Es wird jedoch vermutet, dass sie nicht in FPT sind.*

12.1 Parametrisierte Komplexitätstheorie (Exkurs)

Definition 34 (Parametrisierte Reduktion)
Eine parametrisierte Reduktion *von einem parametrisierten Problem* Π *auf ein zweites parametrisiertes Problem* Π' *besteht aus einer Funktion* f*, die einer Instanz* $\mathcal{I}$ *und einem Parameter* k *von* Π *eine Instanz* $\mathcal{I}'$ *von* Π' *zuordnet, sowie Funktionen* $g, h : \mathbb{N} \to \mathbb{N}$ *so, dass folgende Eigenschaften erfüllt sind:*

(i) $f((\mathcal{I}, k))$ *kann in* $\mathrm{O}(h(k) \cdot p(n))$ *berechnet werden, wobei* n *die Eingabegröße von* $\mathcal{I}$ *und* p *ein Polynom ist.*

(ii) $(\mathcal{I}, k)$ *ist eine Ja-Instanz von* Π *ist genau dann, wenn* $(\mathcal{I}', g(k))$ *eine Ja-Instanz zu* Π' *ist.*

Remark 49

(i) Es gibt eine Hierarchie von Komplexitätklassen $W[t]$*, die vermutlich echt ist, und FPT enthält*

$$FPT \subseteq W[1] \subseteq W[2] \cdots$$

(ii) Mithilfe der parametrisierten Reduktion lässt sich ein Vollständigkeitsbegriff definieren, um die schweren Probleme in $W[t]$ *zu identifizieren.*

(iii) INDEPENDENT SET *ist* $W[1]$*-vollständig und* DOMINATING SET *ist* $W[2]$*-vollständig.*

12.2 Grundtechniken zur Entwicklung parametrisierter Algorithmen

In diesem Kapitel sollen zwei Grundtechniken zur Entwicklung parametrisierter Algorithmen vorgestellt werden:

1. *Kernbildung:* Dabei wird die Instanz durch Anwendung verschiedener Regeln auf einen *(schweren) Problemkern* reduziert.

2. *Tiefenbeschränkte Suchbäume:* Dabei wird eine erschöpfende Suche in einem geeigneten Suchbaum mit beschränkter Tiefe durchgeführt.

Idee 1 (zu 1.)
Reduziere Instanz $(\mathcal{I}, k)$ in Laufzeit $\mathrm{O}(p(\langle\mathcal{I}\rangle))$ auf eine äquivalente Instanz $\mathcal{I}'$, so dass die Größe $\langle\mathcal{I}'\rangle$ von $\mathcal{I}'$ nur von k abhängt. Löse $\mathcal{I}'$ anschließend etwa durch erschöpfende Suche.

Example [VERTEX COVER] Um ein Vertex Cover V' für $G = (V, E)$ mit höchstens k Knoten zu bestimmen, kann man sich die folgenden Beobachtungen zunutze machen:

(i) Für alle $v \in V$ ist entweder v selbst oder alle Nachbarn $\mathcal{N}(v)$ in V'.

(ii) Für ein Vertex Cover V' der Größe k gilt

$$\{v \in V \mid |\mathcal{N}(v)| > k\} \subseteq V',$$

d.h. ein solches Vertex Cover enthält alle Knoten mit Grad größer als k.

(iii) Falls $\Delta(G) \leq k$ und $|E| > k^2$ gelten, so hat G kein Vertex Cover mit k oder weniger Knoten. Dabei bezeichne $\Delta(G)$ den Maximalgrad von G.

Der folgende Algorithmus nutzt diese Eigenschaften aus:

```
Function kernelVertexCover($G = (V, E), k$) : Boolean
    while $\exists v \in V$ : degree($v$) $> k$ do
        if $k = 0$ then return false
        $G := G - v$
        $k := k - 1$
    remove isolated nodes
    if $|E| > k^2$ then return false
    return vertexCover($G, k$)
```

Bemerkung zur Korrektheit: Wenn der Graph $G - v$ des Algorithmus kein Vertex Cover der Größe $k - 1$ hat, so hat G kein Vertex Cover der Größe k. Am Ende ist der Maximalgrad k und jeder Knoten eines Vertex Covers kann deswegen höchstens k Kanten überdecken. Es gilt dann also $|E| \leq k^2$. Demnach hat G kein Vertex Cover der Größe k, falls $|E| > k^2$.

Laufzeitanalyse: Wenn vertexCover(G, k) am Ende aufgerufen wird, hat der Graph G maximal k^2 Kanten. Falls die Berechnung von vertexCover(G, k) mit einer Laufzeit in $\mathrm{O}(f(k))$ ausgeführt werden kann, so ist die Gesamtlaufzeit in $\mathrm{O}(nk + f(k))$.

Nach Entfernen aller isolierten Knoten aus G hat der resultierende Graph höchstens $2k^2$ Knoten, da jede der höchstens k^2 Kanten höchstens zwei Knoten zur Knotenmenge beitragen kann. Es gibt somit höchstens $\binom{2k^2}{k}$ Teilmengen mit k Elementen, die als Vertex Cover für G infrage kommen. Diese Anzahl läßt sich durch $(2 \cdot k^2)^k = \mathrm{O}\big(2^k \cdot k^{2k}\big)$ abschätzen. Damit ergibt sich ein Gesamtaufwand in $\mathrm{O}\big(nk + 2^k \cdot k^{2k}\big)$, wenn erschöpfende Suche verwendet wird.

Der Aufwand läßt sich jedoch durch Verwendung eines tiefenbeschränkten Suchbaums auch auf $\mathrm{O}\big(2^k \cdot k^2\big)$ reduzieren. Wenn sich die Größe des Suchbaums verringern lässt, kann man den Faktor 2^k eventuell noch weiter verbessern.

Idee 2 (zu 2.)
Der Aufwand, der durch Verwendung eines tiefenbeschränkten Suchbaumes entsteht, hängt wesentlich von der Struktur des Baumes ab. Im vorangegangenen Beispiel haben wir einen binären Suchbaum verwendet. Damit ergibt sich die Laufzeit durch die Abschätzung

$$T(k) = (n + m) + T(k - 1) + T(k - 1)\ . \tag{12.1}$$

Allgemein können aber auch andere Reduktionsregeln eingesetzt werden:

$$T(k) = (n + m) + T(k - t_1) + \cdots + T(k - t_s)\ . \tag{12.2}$$

Der Vektor $(t_1, \ldots, t_s)$ *heißt Verzweigungsvektor. Im obigen Beispiel ist der Verzweigungsvektor somit der Vektor* $(1, 1)$.

Mithilfe kombinatorischer Mittel (erzeugende Polynome) kann man das asymptotische Wachstum von $T(k)$ aus dem Verzweigungsvektor bestimmen. Für einige Verzweigungsvektoren ist die sich ergebende Basis in Tabelle 12.1 dargestellt.

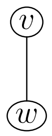
Table 12.1: Basen ausgewählter Verzweigungsvektoren

Verzweigungsvektor	**Basis**
$(1,1)$	2
$(1,2)$	1.618
$(1,3)$	1.4656
$(1,4)$	1.3803
$(1,5)$	1.325
$(2,3)$	1.325
$(3,3)$	1.26
$(3,3,6)$	1.342
$(3,4,6)$	1.305

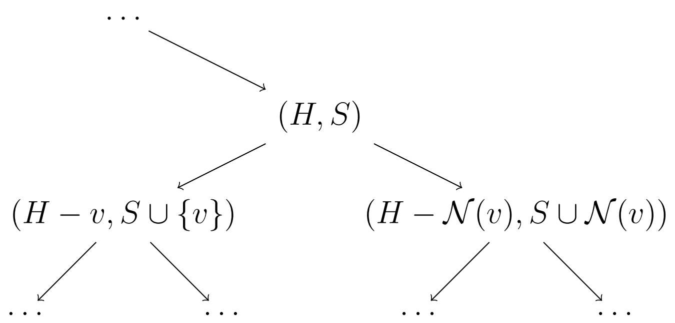
Example Zur Illustration sei hier ein Beispiel mit Verzweigungsvektor $(1,3)$:

```
Function vertexCover2($G = (V,E), k$) : Boolean
    if $|E| = 0$ then return true
    if $k = 0$ then return false
    if $\exists v \in V : \mathsf{degree}(v) = 1$ then
        return vertexCover2($G - \mathsf{neighbor}(v), k - 1$)
    if $\exists v \in V : \mathsf{degree}(v) \geq 3$ then
        return   vertexCover2($G - v, k - 1$)$\vee$
                 vertexCover2($G - \mathcal{N}(v), k - |\mathcal{N}(v)|$)
    assert all nodes have degree 2
    return vertexCoverCollectionOfCycles($G, k$)
```

Gibt es einen Knoten mit Grad 1, so wird keine Verzweigung durchgeführt. Stattdessen wird der Nachbarknoten w von v in das Vertex Cover übernommen.



Gibt es einen Knoten v mit $\mathsf{degree}(v) \geq 3$, kann man verzweigen, indem man entweder v oder alle Nachbarn von v zum Vertex Cover hinzunimmt:



Ansonsten haben alle Knoten Grad 2. Der Graph besteht also aus disjunkten Kreisen. Sei $\langle v_1, v_2, \ldots, v_{q-1}, v_q = v_1 \rangle$ ein solcher Kreis. Die Knoten v_i mit ungeradem Index i bilden einen optimalen Vertex Cover. Dieser lässt sich in linearer Zeit berechnen.

Damit ergibt sich eine Rekursionsgleichung der Form

$$T(k) = (n + m) + T(k - 1) + T(k - 3) \tag{12.3}$$

und daher eine Laufzeit von $\mathrm{O}\big((n + m)1.4656^k\big)$ (siehe Tabelle 12.1).

Weitere Verbesserungen

- Crown Reductions [18]: Sei I Independent Set und $\mathcal{N}(I) = \{v \in V \mid \exists u \in I : \{u, v\} \in E\}$ die Nachbarschaft von I. Sei M ein Matching maximaler Größe im bipartiten Graph $(I \cup \mathcal{N}(I), \{\{u, v\} \in E \mid u \in I, v \in \mathcal{N}(I)\})$. $|M| = |\mathcal{N}(I)| \Rightarrow \exists$ min. Vertex Cover, das $\mathcal{N}(I)$ enthält. Ein geeignetes Independent Set lässt sich mittels Max. Cardinality Matching finden.

- Kerne der Größe $2k$ (wieder mittels Matching-Algorithmen)

- Reduziere Zeit pro rekursivem Aufruf auf $\mathrm{O}(1)$

- Detaillierte Fallunterscheidungen $\rightsquigarrow$ kleinere Konstante im exponentiellen Teil

Dadurch ergibt sich ein Algorithmus mit Laufzeit $\mathrm{O}\big(1.2738^k + kn\big)$ [16].

12.3 Literaturhinweise

Dieses Kapitel stammt teilweise aus dem Skript zur Vorlesung Algorithmentechnik von Prof. Dorothea Wagner vom Wintersemester 09/10. Die algorithmischen

Aspekte dieses Kapitels wie Kernbildung und beschränkte Suchbäume werden in einem einführenden Buch zu Parametrisierten Algorithmen von Niedermeier [74] beschrieben. Die Bücher von Downey und Fellows [73] sowie Flum und Grohe [43] behandeln hauptsächlich parametrisierte Komplexitätstheorie.

Chapter 13

Online Algorithms

Online algorithms are a class of algorithms which process their input in a serial fashion, where future information is only *revealed in parts*. The algorithms must fully process each part and produce a partial output before receiving further input. No information is known about the further input sequence. Therefore, online algorithms are restricted to making decisions based solely on past events. The challenge is to design algorithms that perform well under this lack of knowledge about the future.

In contrast, many algorithms previously developed in these course notes require the whole input to be available at the beginning and do extensive computation on the entirety of the problem data. These are called *offline algorithms*.

However, many practical problems have an intrinsic online nature, for example:

- Paging in a virtual memory system.
- Renting skis vs. buying skis.
- Scheduling jobs in a multi-user system.
- Routing in communication networks.
- Google placing advertisements on websites.

Furthermore, some of the well-known algorithms are online algorithms: insertion-sort can sort an input sequence that is revealed only one item at a time.

Note that online algorithms do not make particular probabilistic assumptions on the unknown input. With such assumptions many of the examples above are typical applications of classic queueing theory.

Most of this chapter is based on the text book [11].

13.1 Competitive Analysis

Prior to focusing on two of these examples we will develop a method to compare different online algorithms. Their quality is determined by ***competitive analysis*** against an optimal offline algorithm. Thus this analysis quantifies the error an online algorithm makes due to lack of information.

Definition 35
Given an optimization problem with input I, for which an offline algorithm OPT *can compute an optimal solution using a cost measure* $\mathrm{OPT}(I)$*, then an online algorithm* ALG *is called* ***c-competitive*** *if there is a constant α such that for all finite input sequences I*

$$\mathrm{ALG}(I) \leq c \cdot \mathrm{OPT}(I) + \alpha$$

where $\mathrm{ALG}(I)$ *is the cost of computing input I using* ALG.

If moreover the additive constant $\alpha \leq 0$, then one says ALG *is* ***strictly c-competitive***

ALG *is strictly c-competitive for*

$$c = \sup_I \frac{\mathrm{ALG}(I)}{\mathrm{OPT}(I)}.$$

The ratio c is also called the ***competitive ratio*** *of* ALG.

Note that c is a worst-case performance measure, depending on the input sequence with worst relative performance.

An algorithm is simply called **competitive** if it is c-competitive for some real constant c. Note that if ALG is c-competitive, then it is also c'-competitive for all real constants $c' > c$. The goal is to find online algorithms with minimal c.

13.1.1 Example: Ski Rental

We now exemplify this analysis method on a typical online problem: ski rental.

Example Say you are planning to spend your winter holiday at a cost-efficient hotel in the Alps near Ischgl and do not own a pair of skis. You have the choice of buying skis for 300 euros from your favorite sports equipment store at home, or to rent skis for a total of 50 euros in Ischgl. Considering that you do not know how often you will go skiing, should you rent or buy skis?

As a first simple online algorithm we suggest buying skis on the sixth trip. Using competitive analysis we can calculate the competitive ratio for this algorithm:

- If you make five or fewer skiing trips, you will pay the optimal cost of 50 euro per trip and thus the ratio to an optimal offline algorithm is $\frac{5 \cdot 50}{5 \cdot 50} = 1$ for these inputs.

- If you make six or more skiing trips, you pay $250 + 300 = 550$ euros in total. However, an optimal offline algorithm can take advantage of its information about the future and pays only 300 euro. For these inputs the ratio is $\frac{550}{300} = \frac{11}{6}$, thus our simple algorithm is $\frac{11}{6}$-competitive.

We now show that no better online algorithm can be devised for this problem.

- Suppose we buy skis earlier, say on the trip $x < 6$. For fewer than x trips, the ratio stays 1, but for exactly x trips an optimal algorithm pays $50\,x$ euros while the online algorithm pays $50(x-1) + 300$. Thus it has a competitive ratio of
$$\frac{50x + 250}{50x} = 1 + \frac{5}{x} \geq 2$$

- Suppose we buy skis later, say on the trip $x > 6$. For exactly x trips an optimal offline algorithm pays only 300 euros while the online algorithm pays again $50(x-1) + 300$ euro. The competitive ratio in this case is
$$\frac{50x + 250}{300} = \frac{5 + x}{6} \geq 2$$

So both buying times $x < 6$ and $x > 6$ yield an algorithm with a competitive ratio greater than 2. All other algorithms, like renting skis while already possessing a pair or buying two pairs, are obviously inferior and thus buying after 6 trips is the optimal algorithm in absence of complete knowledge.

The ski rental example is prototypical of a class of problems in which you must decide whether to pay smaller repeating costs or a large one-time cost, which reduces or eliminates the repeating costs. Other examples are whether to buy a BahnCard or monthly tickets for a public transport system. The idea behind optimal online algorithms for these problems is to not pay the large one-time cost until you would have paid the same amount in smaller repeating costs.

13.2 Paging – Deterministic Algorithms

The paging problem is an example of a well-studied set of online algorithms for which optimal deterministic and randomized algorithms are known. In this section the paging problem is formalized and five deterministic online algorithms for this problem are compared via competitive analysis. Randomized online algorithms for paging are discussed in the next chapter.

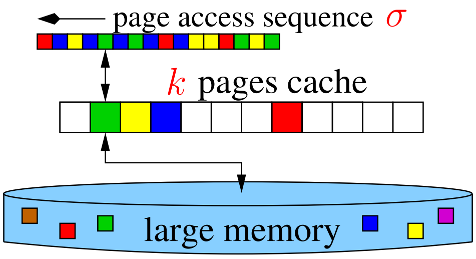


Figure 13.1: Paging in a virtual memory system

The Paging Problem Consider a two-level virtual memory system as in figure 13.1: the system consists of a large main memory storage to which access is relatively slow. To speed up processing a faster cache is used to temporarily store accessed memory locations. Transfer between both memory stores is possible only using fixed-sized memory blocks called ***pages***. The slow main memory $P = \{p_1, \ldots, p_N\}$ contains N pages of which any k-subset can be stored in the cache where $k < N$. A program specifies a sequence of page accesses $\sigma : \{1, \ldots, n\} \to \{1, \ldots, N\}$ of finite length $|\sigma| := n$. Given a request for a page $p_{\sigma(i)}$, the virtual memory system must make $p_{\sigma(i)}$ available in the fast memory. If $p_{\sigma(i)}$ is already loaded in the fast memory we call this a *cache hit*, otherwise a ***page fault*** or *cache miss* occurs and the page must be transferred from main memory to cache memory. Due to the limited size of fast memory, the *paging algorithm* must decide which of the cached pages is to be replaced (*evicted* from cache). Goal of the paging algorithm must be to minimize the number of cache misses.

The memory systems of today's computer architectures are composed of a *multi-level virtual memory system* containing CPU registers, three cache levels on the CPU (L1, L2 and L3), physical RAM and swap partitions on hard-drives. For our discussion on online algorithms we will use the high-level abstract model and disregard technical details like k-associative caches or page sizes. We assume a cache size of k pages, allow cache hits to be free (very fast cache) and penalize a cache miss with cost 1.

Paging is a natural online problem as future memory access patterns are generally not known in advance. Table 13.1 contains the most basic and well-known deterministic paging algorithms.

algorithm		strategy: replace page ...
FIFO	First In / First Out	which was longest in cache
LIFO	Last In / First Out	which was newest in cache
LFU	Least Frequently Used	requested least number of times
LRU	Least Recently Used	with earliest most recent request
FWF	Flush When Full	evict all pages when cache full
LFD	Longest Forward Distance	requested farthest in the future

Table 13.1: Deterministic paging algorithms

All algorithms given above except LFD are online algorithms. LFD is an offline algorithm because it requires knowledge of the future page access order. Furthermore, all these algorithms except FWF are called *demand paging*: they never evict a page unless there is a page fault. One can show that any page replacement algorithm can be modified to be demand paging without adding a cost penalty. For FWF this leads to an algorithm which evicts an arbitrary page when the cache is full.

Figure 13.2 shows a snapshot of LRU and LFD working on an example page request sequence. The current cache, with $k = 3$ in both examples, is depicted in the second row. Conceptually, LRU keeps a backwards reference to the last time each cached page was used. In the current state shown, the red page will be evicted to make room for the requested blue page, as the red page has been longest in cache, relative to the most recent access. LFD, on the over hand, is an offline algorithm and can therefore make use of the future request sequence. Each page in the cache references its next access position. When a page must be evicted, LFD can pick the page which has its next request farthest in the future. In the example, access to the green page causes a page to be evicted and because the yellow one will be requested latest of all in the cache, LFD will remove the yellow one.

A further interesting topic is how efficiently the replacement strategies can be implemented. Simplicity is a key factor for paging algorithms because long calculations would impact the fast memory's access performance on cache misses. However, for competitive analysis of the algorithms with page faults as cost metric, the implementation complexity is immaterial.

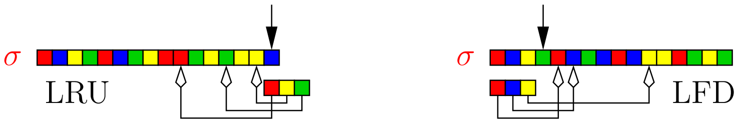


Figure 13.2: Example of LRU and LFD paging algorithm

13.2.1 Longest Forward Distance

Competitive analysis needs an optimal offline algorithm OPT. In the case of paging algorithms this requirement can not only be met, but the optimal page eviction sequence can also be calculated easily.

Theorem 50
LFD *is an optimal offline algorithm for paging.*

Proof.
We prove this theorem by showing that any optimal offline algorithm OPT can be modified to act like LFD on any given page request sequence σ without increasing the number of page faults.

This is shown via inductive application of the following claim: given any paging algorithm ALG and a page request sequence σ, we can construct an offline algorithm ALG'_i for any $i = 1, \dots, |\sigma|$ so that following three properties are satisfied:

(i) ALG'_i processes the first $i-1$ paging requests exactly as ALG does.

(ii) If the i-th request causes a page fault, then ALG'_i evicts the page with *longest forward distance* from cache memory. So for the i-th request the new algorithm behaves the same as LFD.

(iii) The cost in page faults of ALG'_i does not increase: $\mathrm{ALG}'_i(\sigma) \leq \mathrm{ALG}(\sigma)$.

If this claim holds, the theorem can be proven by applying it $n = |\sigma|$ times to any optimal offline algorithm OPT. Start by applying the claim to OPT with $i = 1$ to obtain OPT'_1, which acts like LFD on the first request. Then apply the claim to OPT'_1 with $i = 2$ to obtain OPT''_2 and so on. After n steps the algorithm $\mathrm{OPT}^{(n)}_n$ acts identically to LFD on the request sequence σ. The number of page fault of $\mathrm{OPT}^{(n)}_n$ has not increased due to (iii), nor can it have decreased as OPT was already optimal.

To finish the proof, it remains to show the claim. Given a paging algorithm ALG, we first construct ALG'_i. Figure 13.3 shows an illustration of this construction.

For the first $i-1$ requests, both algorithms must behave identically. Therefore, after the $(i-1)$-th request the cached page sets are identical. Only after handling the i-th request for page x these sets might differ, as ALG'_i will pick the page a with *longest forward distance* to evict. We denote the cached page set of ALG after i requests by $X \cup \{a\}$ and of ALG'_i by $X \cup \{b\}$, where X is the common page set of size $k-1$. If $a = b$, then there was no page fault or ALG already behaves like ALG'_i for the i-th page request and all further page requests can be handled

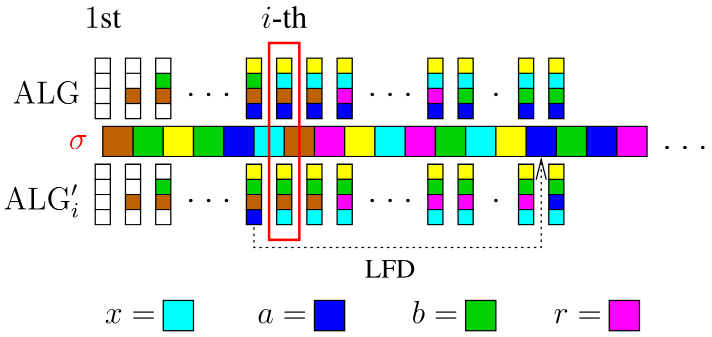


Figure 13.3: Example showing cached page sets of ALG and ALG'_i

identically. We therefore assume $a \neq b$. For the requests following the i-th, ALG'_i shall mimic ALG, except when either page a or b is requested. This strategy is possible, because both algorithms' page sets share $k-1$ pages. If at some point ALG evicts a, then ALG'_i can evict b instead, and from there on both are identical. Otherwise b will be requested before a, as a has longest forward distance. When b is requested, ALG must evict a page r to make room for b, while ALG'_i still has b in cache. If $r = a$, then both page sets are equal from this point on. Otherwise ALG now has $\tilde{X} \cup \{a\}$ cached while ALG'_i has $\tilde{X} \cup \{r\}$ with a common page set $\tilde{X}$. Again ALG'_i can continue like ALG, until eventually a is requested. At this time ALG'_i can evict r to make room for a, after which both page sets are identical.

The constructed algorithm ALG'_i satisfies conditions (i) and (ii). The difficulty lies in showing (iii): ALG'_i may not suffer more page faults than ALG. An additional page fault can only occur before both page sets are identical again. Both algorithms behave the same except for requests to pages a and b, so both incur the same page faults for access to other pages. If $a \neq b$, then ALG suffers a page fault when b is accessed before a. ALG then evicts r, while ALG'_i still holds r. The page r then takes the role of b. This can happen multiple times further increasing ALG's cost. Eventually a is requested, for which ALG'_i suffers a page fault and can then remove r. This page fault is covered by the previous one when r was evicted by ALG. Thus ALG'_i never suffers more page faults than ALG. □

13.2.2 Competitive Paging Algorithms

We now have an optimal paging algorithm $\mathrm{OPT} = \mathrm{LFD}$. Of course, LFD is an offline algorithm and we now try to find the best online algorithm by competitive analysis against LFD.

Of the previously represented online algorithms, two can immediately be excluded: LIFO and LFU are not competitive, as we can show that their competitive ratio c can grow infinitely large.

Let k be the cache size and $P = \{p_1, \dots, p_k, p_{k+1}\}$ the pages in the slow main memory. When processing the request sequence

$$\sigma = p_1, p_2, \dots, p_k, (p_{k+1}, p_k)^m$$

for some integer m, LIFO will first cache pages $p_1, \dots, p_k$. For the $(k+1)$-st request LIFO will evict p_k to make room for p_{k+1}, which it will remove again on the $(k+2)$-nd request and so on. Thus LIFO will incur page faults on every request after the k-th one, in total $2m$, whereas LFD only suffers $k+1$ page faults in total. We can therefore provide for every constant c a request sequence of length $|\sigma| > k + c(k+1)$, which violates definition 35.

Again let k be the cache size and $P = \{p_1, \dots, p_k, p_{k+1}\}$ the pages in the slow main memory. For every integer m we can consider the request sequence

$$\sigma = p_1^m, p_2^m, \dots, p_{k-1}^m, (p_k, p_{k+1})^m$$

LFU will load the pages $p_1, \dots, p_{k-1}$ for the first $k-1$ page faults. After the first $(k-1)m$ requests however, LFU will suffer a page fault for each of the $2m$ following requests. For the first page fault p_k, the cache still has one empty slot, but following this each request forces LFU to evict a page: p_k when p_{k+1} is requested and vice versa. Since m can be made arbitrarily large and LFD requires only $k+1$ page faults, the competitive ratio of LFU is unbounded.

Following this sobering result, we show another theorem, which establishes a lower bound on the competitive ratio for *any* online paging algorithm.

Theorem 51
No deterministic online paging algorithm can achieve a better competitive ratio than k, where k is the cache size.

Proof.
Let ALG be an online paging algorithm working on a cache size of k. We show there is an arbitrarily long request sequence σ for which

$$|\sigma| = \text{ALG}(\sigma) \geq k \cdot \text{LFD}(\sigma)$$

Assume there are $k+1$ pages in slow memory and that the cache already holds k pages. Thus there is always a page not in the cache. Since ALG is deterministic, a malicious page request sequence can be constructed for which each request results in a page fault: in each step a page not in cache is requested. For such a sequence $\text{ALG}(\sigma) = |\sigma|$, whereas LFD incurs at most one page fault on every k requests: i.e. $\text{LFD}(\sigma) \leq \frac{|\sigma|}{k}$. This is due to LFD being able to have the next k page requests in cache for each position in the sequence. Thus $\text{ALG}(\sigma) \geq k \cdot \text{LFD}(\sigma)$ or $\frac{\text{ALG}(\sigma)}{\text{LFD}(\sigma)} \geq k$. □

13.2.3 Conservative Paging Algorithms

For the continued analysis of online paging algorithms in this chapter, we now refine the paging problem to allow different sizes of cache memory allotted to the online and optimal offline algorithm. In the ***(h, k)-paging problem*** we compare an optimal offline algorithm working on h cache pages to an online algorithm with cache $k \geq h$. With $h = k$ we still include the equal cache size scenario, while for $h < k$ we give the online algorithm the advantage of more cache memory as compensation for its lack of knowledge. This approach may lead to more realistic competitive ratios.

Note that some paging algorithms incur what is known as ***Bélády's anomaly***: on some request sequences these algorithms suffer fewer page faults with a smaller cache size. FIFO for example is prone to this anomaly. It shows that the effects of different cache sizes can be subtle.

To conduct competitive analysis of a broader range of paging algorithms we define the following class:

Definition 36
A paging algorithm is called **conservative** *if it incurs at most k page faults on any request sequence containing at most k distinct pages.*

LRU, FIFO and even FWF are conservative. LFU and LIFO are not conservative, they are not even competitive. The page requests sequences in the proof of non-competitiveness are good starting points for the reader to show this observation.

We now show that all conservative paging algorithms have at least the following competitive ratio.

Theorem 52
Any conservative online algorithm ALG*, which works on a cache of size k, is $\frac{k}{k-h+1}$-competitive to any optimal offline algorithm* OPT *with $h \leq k$ pages of cache.*

Proof.
Let σ be a fixed page request sequence. We first partition this sequence into phases: phase 0 is the empty sequence and for $i > 0$ let phase i be the longest sequence of page requests following phase $i - 1$ that contains at most k distinct pages. Such a

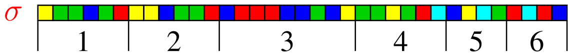


Figure 13.4: The 4-phase partition of an example sequence

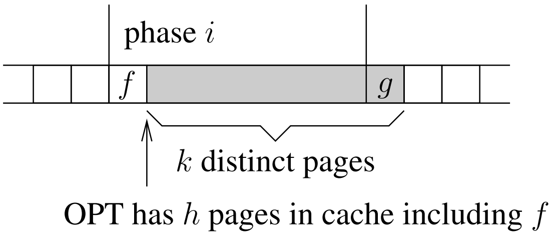


Figure 13.5: Illustration of the page faults of OPT

partition is well-defined and called a ***k-phase partition*** of σ. It does not depend on any particular algorithm. (see figure 13.4 for an example)

The k-phase partition is constructed so that the definition of a conservative paging algorithm can easily be applied: in each sub-sequence a conservative algorithm will have at most k page faults.

Now consider how many page faults an optimal offline paging algorithm OPT working with $h \leq k$ cache pages can have in a k-phase (Figure 13.5 shows an illustration). Take any phase $i \geq 1$, then name the first page f and regard the input sequence following f up to the first request of phase $i+1$, which is labeled g if it exists. An optimal paging algorithm using h cache pages will have $h-1$ pages other than f in fast memory. However, since the input sequence following f contains k distinct pages, at least $k-(h-1)$ pages are missing from cache memory. Therefore at least $k-h+1$ unavoidable page faults will occur even to OPT in the sequence following f up to and including g. If the last phase contains only $k' < k$ distinct page requests, the argument shows that at least $k'-h+1$ page faults occur for OPT. For competitive analysis this can be ignored, as it is only an additive constant cost.

So during each k-phase the competitive online algorithm ALG suffers at most k page faults, while for OPT at least $k-h+1$ page faults occur. Therefore, we showed the following inequality for every input sequence σ

$$\mathrm{ALG}(\sigma) \leq \frac{k}{k-h+1} \cdot \mathrm{OPT}(\sigma) + r$$

where $r \leq k$ is the maximum number of page faults of ALG for the last phase. □

We can apply this theorem to some example problems: for (k,k)-paging every conservative online algorithm, like FIFO, LRU or FWF, has a competitive ratio of k, which we showed in theorem 51 to be a lower bound for this problem. These online algorithms are therefore optimal in this context.

Applying theorem 52 with $h = \frac{k}{2}$ shows that conservative online algorithms are 2-competitive for the $(h, 2h)$-paging problem. The lower bound of k only applies to the special case (k,k)-paging.

13.3 Randomized Online Algorithms

In this chapter we will explore how to use randomization to avoid the lower bound of k for (k, k)-paging.

Randomized algorithms are allowed to make *random choices* in their online decision process. For analysis of these algorithms we view the problem as a game between an ***online player***, the online algorithm, and a malicious ***adversary***. This adversary attempts to pick a page request sequence which maximizes the competitive ratio. In the deterministic case there was just one natural adversary model, which knows both the algorithm and its current cache, and from these it could easily determine which request would lead to a page fault (theorem 51).

For analysis of randomized algorithms we must formalize which information is available to an adversary. All adversary models assume that the algorithm itself, the current page set and the probability distributions from which the algorithm draws random choices are known. The following models distinguish between how much of the drawn random variates is known in advance and when the page request sequence must be fixed.

- An ***oblivious adversary*** does not know the random choices and must choose the entire request sequence in advance. This is the weakest adversary as it can only exploit a-priori knowledge about the algorithm.

- An ***adaptive-online adversary*** is allowed to determine the next page request by inspecting all random choices previously made by the online algorithm. This models an input-output scenario where the adversary receives the output and can then devise new malicious inputs. The input requests are determined *online*, without knowing the future random choices.

- An ***adaptive-offline adversary*** knows the outcome of all random choices the online algorithm is going to make in advance and can determine the page request sequence offline beforehand.

These three adversary models are also sometimes called *weak*, *medium* and *strong* models. Randomization does not help against adaptive-offline adversaries: one can show that if there exists a c-competitive randomized online algorithm against an adaptive-offline adversary, then there also exists a deterministic c-competitive online algorithm for the same problem.

We will focus on analysis against an oblivious adversary. Since this adversary must define the page request sequence σ in advance, we can assign it the same cost $\mathrm{OPT}(\sigma)$ as an optimal deterministic paging algorithm requires on the sequence. To rank a randomized online algorithm ALG against an oblivious adversary, we need to regard $\mathrm{ALG}(\sigma)$ as a random variable of the possible random choices that the algorithm makes. We therefore provide

Definition 37
A randomized online algorithm ALG *is called* ***c-competitive*** *against an oblivious adversary if there is a constant* α *such that for all finite input sequences* σ

$$\mathbf{E}(\mathrm{ALG}(\sigma)) \leq c \cdot \mathrm{OPT}(\sigma) + \alpha$$

where $\mathbf{E}(\mathrm{ALG}(\sigma))$ *is the expected value of the random variable* $\mathrm{ALG}(\sigma)$ *over all possible random choices in* ALG.

Note that the randomized competitive ratio is not defined as a worst-case measure over *all random choices* in ALG, but is defined as the worst case over *all input sequences*. For the randomized competitive ratio one needs to finds the input sequence with the worst *expected cost* of ALG over all possible random choices. For the oblivious adversary, the input sequence does not depend on the random choices of ALG and therefore $\mathrm{OPT}(\sigma)$ can be fixed independently of ALG's choices.

13.3.1 Randomized Marking Algorithms

We now distinguish a special class of online paging algorithms called *marking algorithms*, which includes a specific randomized paging algorithm that we will analyze against an oblivious adversary.

Definition 38
We call a paging algorithm a ***marking algorithm****, if its operations can formally be described using a one bit mark for each page of the slow memory. Initially, with an empty cache, all pages are unmarked. On request of a page, this page is marked and loaded into cache memory, if it is not already there. Only unmarked pages may be evicted if the algorithm needs to removed a page from cache. If all pages are marked and there is a page fault, then all marked pages are unmarked and one of them is evicted.*

The marking/unmarking phases of this definition are similar but not identical to the k-phase partition used in the proof of theorem 52. Marking algorithms can only differ in the decision, which of the unmarked pages is evicted. FWF is maybe the most primitive marking algorithm, as it just evicts all unmarked pages on a page fault requiring page eviction. More elaborately, LRU is also a marking algorithm and it picks the unmarked page least recently used. FIFO is not a marking algorithm.

For competitive analysis we target RMARK, which is the marking algorithm that evicts an unmarked page chosen uniformly at random when required.

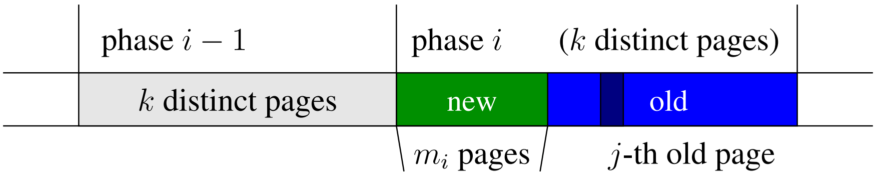


Figure 13.6: Worst-case for RMARK: new and old pages in a k-phase partition

In context of the following analysis we need a common definition: for any $k \in \mathbb{N}_1$ the ***k*-th harmonic number H_k** is defined as

$$H_k = \sum_{i=1}^{k} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k}$$

The harmonic number can be bounded by $\ln k < H_k \leq 1 + \ln k$ for all $k \geq 1$.

Theorem 53
The randomized paging algorithm RMARK *is* $(2H_k)$*-competitive against an oblivious adversary.*

Proof.
We again look at the k-phase partition of a fixed request sequence σ and analyze the expected number of page faults in phase i for RMARK. At the beginning of phase i there are k pages in cache as requested in phase $i-1$. We call these pages *old*, whereas pages that are requested in phase i, but not in phase $i-1$, are labeled as *new*. Every page in phase i is either old or new, let m_i be the number of new pages. Access to each of the m_i new pages must result in a page fault, whereas old pages may still be in the cache from the previous phase. Consecutive accesses to the same old or new pages do not incur any further page faults and we can therefore concentrate on distinct page requests. The worst possible order of page requests for the online algorithm is first all new pages followed by the old pages (see figure 13.6), because each new page request evicts an old page which might be used later. The online algorithm, however, cannot take into account which of the old pages will be requested.

So there are m_i unavoidable page faults for the new pages, which are marked and therefore cannot be evicted in this phase. We need to determine the expected value of page faults for the $k-m_i$ old pages. For this consider the probability that the j-th old page is still in the cache when it is first requested: since it is unmarked it must be one of the $k-m_i-(j-1)$ unmarked old pages *in the cache* (there are only $k-m_i$ slots for old pages and $j-1$ are already taken). Dividing this by the total number of unmarked old pages $k-(j-1)$ in this request sequence (including those not requested) yields the probability of a cache hit for the j-th old page. So, the probability for the j-th old page to cause a page fault is

$$1 - \frac{k - m_i - (j-1)}{k - (j-1)} = \frac{m_i}{k - (j-1)}$$

To get the expected number of faults in phase i, we sum over all old pages:

$$\begin{aligned} m_i + \sum_{j=1}^{k-m_i} \frac{m_i}{k-j+1} &= m_i + m_i \left(\frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{m_i+1} \right) \\ &= m_i + m_i(H_k - H_{m_i}) = m_i(H_k - H_{m_i} + 1) \leq m_i H_k \end{aligned}$$

If we say the request sequence σ has N phases, then we get

$$\mathbf{E}(\mathrm{RMARK}(\sigma)) \leq H_k \sum_{i=1}^{N} m_i$$

We now have an upper bound for the cost of RMARK, to complete the competitive analysis we require a lower bound on $\mathrm{OPT}(\sigma)$. Here we need to look at two adjacent phases $i-1$ and i. Since there are m_i new pages in phase i, there must be at least $k + m_i$ distinct pages in the request sequence consisting of phases $i-1$ and i. Thus in phase $i-1$ and i even OPT must incur at least m_i page faults. Hence, we have the following lower bound.

$$\mathrm{OPT}(\sigma) \geq \frac{1}{2} \sum_{i=1}^{N} m_i$$

By dividing the two bounds we can calculate a randomized competitive ratio of RMARK:

$$\frac{\mathbf{E}(\mathrm{RMARK}(\sigma))}{\mathrm{OPT}(\sigma)} \leq \frac{H_k \sum_{i=1}^{N} m_i}{\frac{1}{2} \sum_{i=1}^{N} m_i} = 2H_k$$

□

So with RMARK we have a simple randomized algorithm with a competitive ratio of $2H_k$. Note that H_k is usually a lot smaller than k (for example $H_8 \approx 2.72$), therefore even this crude randomized algorithm can be better in practice than LRU or FIFO, and probably is even easier to implement.

13.3.2 Lower Bound for Randomized Paging Algorithms

We now consider whether it is possible to design a randomized paging algorithm, that has a better competitive ratio than RMARK. This is indeed possible but more complicated. Instead we show a lower bound for the competitive ratio for any randomized paging algorithm.

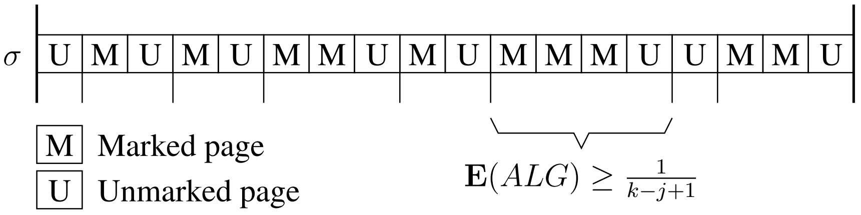

σ
U M U M U M M U M U M M M U U M M U
M Marked page
U Unmarked page
$\mathbf{E}(ALG) \geq \frac{1}{k-j+1}$

Figure 13.7: Eight sub-phases of a (super-)phase each containing some marked pages and one new unmarked page

Theorem 54

No randomized online paging algorithm working with cache size k and slow memory size $N \geq k + 1$ can have a better competitive ratio than H_k against an oblivious adversary.

Proof.

Let ALG be a randomized online algorithm with k cache pages and $N = k + 1$ pages in slow memory. Since an oblivious adversary must construct the request sequence in advance, we must devise a method which yields a sequence that results in a competitive ratio of H_k for any ALG, then the competitive ratio, as a worst-case measure over all sequences, is greater for all possible algorithms. The adversary knows about the algorithm's randomization mechanism and can therefore maintain a probability vector $(p_1, \ldots, p_N)$, where p_i is the probability for page i of the slow memory to not be in the current cache set. This vector can be calculated by the adversary offline using the probability distributions in ALG for each position in the request sequence iteratively, while $\sum_{i=1}^{N} p_i = 1$ holds in each step. Thus the expected cost for a request to page i is p_i.

We continue to construct a malicious sequence based on the k-phase partition schema used in the proof to theorem 52. In each phase ALG will suffer at least H_k page faults, while OPT can have $k - 1$ pages in cache and incurs only 1 page fault. This then results in a competitive ratio of at least $\frac{H_k}{1}$. To construct the phases, the adversary uses a "reverse" marking algorithm and thus must store the set of currently marked pages.

Each phase starts with all pages unmarked. During the phase all k pages will be accessed at least once and thereby marked. The trick is to use the probability vector to increase the expected number of page faults between two accesses to unmarked pages. This is possible because there are $k + 1$ pages in total.

Each k-phase is composed of k sub-phases, which in turn are composed of zero or more requests to marked pages already in cache and terminate with one request to an unmarked page, which may or may not be in cache (see figure 13.7). At the start of the j-th sub-phase there are exactly $k - (j - 1)$ unmarked pages. As each sub-phase increases the number of marked pages by one, thus at the end of the

k-th sub-phase all will become unmarked again and a new (super-)phase begins with one marked and k unmarked pages ($N = k + 1$). If we can ensure that the expected cost of ALG for sub-phase j is $\frac{1}{k-j+1}$, then the total expected cost for phase i would be

$$\sum_{j=1}^{k} \frac{1}{k-j+1} = \frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{1} = H_k \tag{13.1}$$

At the start of the j-th sub-phase the set of marked pages M has cardinality j, while the set of unmarked pages U contains $(k+1) - j$ items. Consider $\gamma := \sum_{i \in M} p_i$, which is the total probability of any marked page not being in the cache. If $\gamma = 0$, there is an unmarked page $a \in U$ with $p_a \geq \frac{1}{k+1-j}$ as $|U| = k+1-j$. By requesting this page the expected cost can be met and the sub-phase ended. Otherwise with $\gamma > 0$, there exists a marked page m with $p_m > 0$. Let $\varepsilon := p_m$ and request this page as the first in the sub-phase. Then generate subsequent requests to marked pages using following loop:

> While the expected cost for ALG is less than $\frac{1}{k+1-j}$ and $\gamma > \varepsilon$, request the marked page x with maximal probability p_x.

The expected cost for ALG increases with each iteration of the loop as $p_x > 0$, so the loop must terminate. (Actually, each additional request contributes at least $\frac{\gamma}{|M|} > \frac{\varepsilon}{|M|} > 0$ cost, because $\gamma > \varepsilon$.) If the loop terminates with expected cost $\geq \frac{1}{k+1-j}$, then the adversary can pick any unmarked page to end the sub-phase. If, however, the loop terminates with expected cost $< \frac{1}{k+1-j}$, then $\gamma \leq \varepsilon$ must have been the reason. In this case the adversary must request the unmarked page b with maximal p_b, for which $p_b \geq \frac{1-\gamma}{|U|} = \frac{1-\gamma}{k+1-j}$. The total expected cost for ALG is in this case $\varepsilon + p_b$, which is

$$\varepsilon + p_b \geq \varepsilon + \frac{1-\gamma}{k+1-j} \geq \varepsilon + \frac{1-\varepsilon}{k+1-j} \geq \frac{1}{k+1-j}$$

Thus the total expected cost for any sub-phase is $\geq \frac{1}{k+1-j}$ and therefore for the whole phase at least H_k (due to eq. 13.1). The variable ε is used as an arbitrary bound to the number of marked pages requested.

In each phase the expected cost for ALG is at least H_k, while the cost for OPT is at most 1. Thus the competitive ratio of ALG is at least H_k. Since we did not make any assumption about how ALG works on the page request sequence, this lower bound holds for all randomized online paging algorithms. □

This closes these chapters on competitive analysis of online algorithms. We have shown many analytic results that tend to be very pessimistic about competitive performance. But keep in mind that this performance is always against a much more powerful, optimal offline algorithm. The results of these optimal algorithms are often also not easy to come by. In practice the best paging algorithms are LRU and RMARK (see figure 8.1 in [11]).

In spite of these disadvantages, competitive analysis is a solid foundation to give guarantees on the worst-case performance of online algorithms.

Bibliography

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] A. V. Aho, D. S. Johnson, R. M. Karp, S. Rao Kosaraju, C. C. McGeoch, C. H. Papadimitriou, and P. Pevzner. Emerging opportunities for theoretical computer science. *SIGACT News*, 28(3):65–74, 1997.

[3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[4] E. Althaus and K. Mehlhorn. Traveling salesman-based curve reconstruction in polynomial time. *SIAM Journal on Computing*, 31(1):27–66, 2002.

[5] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Implementing the Dantzig-Fulkerson-Johnson algorithm for large traveling salesman problems. *Math. Programming*, 97(1–2):91–153, 2003.

[6] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

[7] R. Beier and B. Vöcking. Typical properties of winners and losers in discrete optimization. In *36th ACM Symposium on the Theory of Computing*, pages 343–352, 2004.

[8] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, and N. Wolpert. EXACUS—efficient and exact algorithms for curves and surfaces. In *13th ESA*, volume 3669 of *LNCS*, pages 155–166, 2005.

[9] T. Beth and M. Clausen, editors. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 4th International Conference, AAECC-4, Karlsruhe, FRG, September 23-26, 1986, Proceedings*, volume 307 of *Lecture Notes in Computer Science*. Springer, 1988.

[10] L Blum, M Blum, and M Shub. A simple unpredictable pseudo random number generator. *SIAM Journal on Computing*, 15:364–383, May 1986.

[11] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, 1998.

[12] F. C. Botelho, R. Pagh, and N. Ziviani. Simple and space-efficient minimal perfect hash functions. In *10th Workshop on Algorithms and Data Structures*, volume 4619 of *LNCS*, pages 139–150. Springer, 2007.

[13] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. In *6th Workshop on Algorithm Engineering and Experiments*, 2004.

[14] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28:1627–1640, May 1999.

[15] C. Burnikel, J. Könemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in leda. In *SCG '95: 11th annual symposium on Computational geometry*, pages 418–419, New York, NY, USA, 1995. ACM.

[16] Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved Parameterized Upper Bounds for Vertex Cover . In *MFCS '06: Proceedings of the 31th International Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, pages 238–249. Springer, 2006.

[17] B. V. Cherkassky and A. V. Goldberg. On implementing push-relabel method for the maximum flow problem. In *IPCO: 4th Integer Programming and Combinatorial Optimization Conference*, 1995.

[18] Benny Chor, Michael R. Fellows, and David Juedes. Linear Kernels in Linear Time, or How to Save k Colors in O(n^2) Steps. In *Proceedings of the 30th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'04)*, Lecture Notes in Computer Science, pages 257–269. Springer, 2004.

[19] Nicos Christofides. Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem. Technical Report Technical Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, 1976.

[20] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Transactions on Computers*, C-33(9):828–834, 1984.

[21] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS State-of-the-Art Survey*, pages 117–139. Springer, 2009.

[22] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard Template Library for XXL data sets. *Software Practice & Experience*, 38(6):589–637, 2008.

[23] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 138–148, San Diego, 2003.

[24] C. Demetrescu, I. Finocchi, G. F., and Italiano. Algorithm engineering. *Bulletin of the EATCS*, 79:48–63, 2003.

[25] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, August 1994.

[26] M. Dietzfelbinger and C. Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *Lecture Notes in Computer Science*, pages 166–178. Springer, 2005.

[27] D.E. Drake and S. Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85(4):211–213, 2003.

[28] Arnold I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5(12):6–9, December 1956. First paper in open literature on hashing. First use of hashing by taking the modulus of division by a prime number. Mentions chaining for collision handling, but not open addressing.

[29] P. Erdös and A. Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[30] R. Fleischer, B. Moret, and E. Meineche Schmidt, editors. *Experimental Algorithmics From Algorithm Design to Robust and Efficient Software*, volume 2547 of *LNCS*. Springer, 2002.

[31] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time. In *20th International Symposium on Theoretical Aspects of Computer Science*, number 2607 in LNCS, pages 271–282, Berlin, 2003. Springer.

[32] V. Kumar G. Karypis. Multilevel k-way partitioning scheme for irregular graph. *J. Parallel Distrib. Comput.*, 48(1), 1998.

[33] E. N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

[34] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):1–15, September 1998.

[35] Ronald L. Graham. Bounds for Certain Multiprocessor Anomalics. *Bell System Technical Journal*, 45:1563–1581, 1966.

[36] Ronald L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[37] C. Gutwenger and P. Mutzel. A linear time implementation of SPQR trees. In J. Marks, editor, *Graph Drawing*, volume 1984 of *LNCS*, pages 77–90. Springer-Verlag, 2001.

[38] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM*, 34(1):144–162, January 1987.

[39] J. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. of the ACM*, 21(4):549–568, 1974.

[40] Juraj Hromkovic. *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*. Texts in Theoretical Computer Science. Springer, 2004.

[41] Oscar H. Ibarra and Chul E. Kim. Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. *Journal of the ACM*, 22(4):463–468, October 1975.

[42] R. E. Tarjan J. E. Hopcroft. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.

[43] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006.

[44] D. S. Johnson. A theoretician's guide to the experimental analysis of algorithms. In M. Goldwasser, D. S. Johnson, and C. C. McGeoch, editors, *Proceedings of the 5th and 6th DIMACS Implementation Challenges*. American Mathematical Society, 2002.

[45] M. Jünger, S. Leipert, and P. Mutzel. A note on computing a maximal planar subgraph using PQ-trees. *IEEE Transactions on Computer-Aided Design*, 17(7):609–612, 1998.

[46] M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications (JGAA)*, 1(1):1–25, 1997.

[47] Hans Kellerer and Ulrich Pferschy. Improved Dynamic Programming in Connection with an FPTAS for the Knapsack Problem . *Journal of Combinatorial Optimization*, 8(1):5–11, 2004.

[48] D. E. Knuth. *The Art of Computer Programming—Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.

[49] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* Addison Wesley, 2003.

[50] Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[51] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. *Computational Geometry*. Springer, 2000.

[52] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8:3–30, January 1998.

[53] Makoto Matsumoto, Takuji Nishimura, Mariko Hagita, and Mutsuo Saito. Cryptographic mersenne twister and fubuki stream/block cipher. Cryptology ePrint Archive, Report 2005/165, 2005. `http://eprint.iacr.org/`.

[54] C. C. McGeoch, D. Precup, and P. R. Cohen. How to find big-oh in your data set (and how not to). In *Advances in Intelligent Data Analysis*, number 1280 in LNCS, pages 41–52, 1997.

[55] C.C. McGeoch and B. M. E. Moret. How to present a paper on experimental work with algorithms. *SIGACT News*, 30(4):85–90, 1999.

[56] K. Mehlhorn. *Data Structures and Algorithms, Vol. I — Sorting and Searching*. EATCS Monographs on Theoretical CS. Springer-Verlag, Berlin/Heidelberg, Germany, 1984.

[57] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996.

[58] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[59] K. Mehlhorn and P. Sanders. Scanning multiple sequences via cache memory. *Algorithmica*, 35(1):75–93, 2003.

[60] K. Mehlhorn and G. Schäfer. Implementation of weighted matchings in general graphs: The power of data structure. *ACM Journal of Experimental Algorithmics*, 7, 2002.

[61] Kurt Mehlhorn and Stefan Näher. *LEDA, A platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[62] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[63] U. Meyer, P. Sanders, and J. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS Tutorial*. Springer, 2003.

[64] U. Meyer, P. Sanders, and J. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS Tutorial*. Springer, 2003.

[65] B. M. E. Moret. Towards a discipline of experimental algorithmics. In *5th DIMACS Challenge*, DIMACS Monograph Series, 2000.

[66] J. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[67] J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945.

[68] Pagh and Rodler. Cuckoo hashing. In *ESA: Annual European Symposium on Algorithms*, volume 2161 of *LNCS*. Springer, 2001.

[69] S. Pettie and P. Sanders. A simpler linear time 2/3-approximation for maximum weight matching. *Information Processing Letters*, 91(6):271–276, 2004.

[70] T. Polzin and S. V. Daneshmand. Extending reduction techniques for the Steiner tree problem. In *10th Eur. Symp. on Algorithms*, volume 2461 of *LNCS*, pages 795–807. Springer, 2002.

[71] K. R. Popper. *Logik der Forschung*. Springer, 1934. English Translation: *The Logic of Scientific Discovery*, Hutchinson, 1959.

[72] Pagh R. Cuckoo hashing for undergraduates. `http://www.it-c.dk/people/pagh/papers/cuckoo-undergrad.pdf`, 2006.

[73] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[74] R. Niedermeier. *Invitation to Fixed Parameter Algorithms*. Oxford University Press, 2006.

[75] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '02, pages 233–242. Society for Industrial and Applied Mathematics, 2002.

[76] P. Sanders. Fast priority queues for cached memory. In *ALENEX '99, Workshop on Algorithm Engineering and Experimentation*, number 1619 in LNCS, pages 312–327. Springer, 1999.

[77] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[78] P. Sanders. Algorithm engineering – an attempt at a definition. In *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2009.

[79] P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In *11th ACM-SIAM Symposium on Discrete Algorithms*, pages 849–858, 2000.

[80] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *13th European Symposium on Algorithms (ESA)*, volume 3669 of *LNCS*, pages 568–597. Springer, 2005.

[81] P. Sanders and S. Winkel. Super scalar sample sort. In *12th European Symposium on Algorithms*, volume 3221 of *LNCS*, pages 784–796. Springer, 2004.

[82] J. Singler, P. Sanders, and F. Putze. MCSTL: The multi-core standard template library. In *13th International Euro-Par Conference*, volume 4641 of *LNCS*, pages 682–694. Springer, 2007.

[83] D. Spielman and S.-H. Teng. Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time. In *33rd ACM Symposium on Theory of Computing*, pages 296–305, 2001.

[84] D. Gollman T. Beth. Algorithm engineering for public key algorithms. *IEEE Journal on Selected Areas in Communications*, 7(4):458–466, 1989.

[85] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[86] J. C. Venter, M. D. Adams, and E. W. Myers et al. The sequence of the human genome. *Science*, 291(5507):1304–1351, 2001.

[87] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[88] Rolf Wanka. *Approximationsalgorithmen: Eine Einführung*. Leitfäden der Informatik. Teubner, 2006.

[89] I. Wegener. Simulated annealing beats metropolis in combinatorial optimization. In *32nd International Colloquium on Automata, Languages and Programming*, pages 589–601, 2005.

[90] Andrew C. Yao. Uniform hashing is optimal. *Journal of the ACM*, 32:687–693, July 1985.