Mathematisches Forschungsinstitut Oberwolfach

Report No. 25/2007

Algorithm Engineering

Organised by
Giuseppe F. Italiano (Roma) Petra Mutzel (Dortmund) Peter Sanders (Karlsruhe)
Martin Skutella (Dortmund)

May 6th – May 12th, 2007

Abstract. Algorithm Engineering is concerned with the design, theoretical analysis, implementation, and experimental evaluation of algorithms. It provides methodologies and tools for designing, developing and engineering efficient algorithms, and aims at bridging the gap between theory and practice in the field of algorithmic research. Algorithm Engineering is by now an emerging discipline, gaining momentum and credibility throughout the whole research community.

Mathematics Subject Classification (2000): 68W01, 68W40, 68Q10, 68Q25.

Introduction by the Organisers

The workshop *Algorithm Engineering*, organized by Giuseppe Italiano (Roma), Petra Mutzel (Dortmund), Peter Sanders (Karlsruhe), and Martin Skutella (Dortmund) was held May 6th – May 12th, 2007. The goal of the workshop was to bring together researchers with different background, e.g., from combinatorial optimization, algorithmic theory, and algorithm engineering, in order to strengthen and foster collaborations in the area of algorithm engineering and to identify key research directions for the future. There were 47 international participants out of which 23 mainly work in the area of algorithm engineering, 11 in algorithm theory, and 13 in combinatorial optimization. A considerable number of participants visited the Oberwolfach Institute (MFO) for the first time.

In five survey talks given by renowned specialists in the field, the state of the art in selected areas of high interest was demonstrated. The introductory talk was presented by Giuseppe Italiano (Roma) who gave "a personal and historical perspective" of algorithm engineering. Bill Cook (Atlanta) presented the newest results as well as engineering efforts for solving huge TSP instances. Friedhelm Meyer auf der Heide (Paderborn) suggested new algorithmic challenges in the area of Computer Graphics. Ian Munro (Waterloo) introduced the audience into the area of succinct data structures which is getting increasingly important with huge data sets. In this context, also external memory and cache-oblivious algorithms are getting growing interest. The survey talk by Gerth Brodal (Aarhus) presented theory and practice for these type of algorithms.

In addition, we organized a series of sessions containing shorter contributions of roughly 30 minutes. Here, we particularly encouraged young researchers to present their recent results. The focus of the program lay on recent developments in various areas that are relevant for the broad topic of algorithm engineering. Examples are mathematical programming, external memory algorithms, succinct data structures, resilient data structures, dynamic data structures, online algorithms, graph algorithms, geometric computation, analysis of local search algorithms, and algorithmic game-theory. In total, we had 32 talks of which 15 can be assigned to the area of algorithm engineering, 8 to the area of algorithmic theory, and 7 to the area of combinatorial optimization, which we think was an ideal mixture for the purpose of this workshop. We found it encouraging that after the talks a lot of discussion took place between the participants.

We also like to mention that Peter Sanders (Karlsruhe) introduced the new DFG Priority Program 1307 *Algorithm Engineering*, which will start in fall 2007. As the coordinator of this program he pointed out the specific goals of the program and the chances for the area of algorithm engineering. All of the participants had the chance to discuss different aspects of algorithm engineering in the open problem session on Thursday evening.

In our view, and as expressed by many of the participants, the workshop was a great success. We (including many participants) have learned a lot during this week, also from the three different research communities present at the workshop, and got many new ideas for exciting research projects. The program of the workshop was well-received by the participants as the good quality of presentations and the enthusiasm of the speakers and the audience gave raise to joint research among the participants. In particular exchange between young researchers and experienced scientists was promoted.

We wish to thank Oberwolfach for this unique opportunity to bring together the three different research groups at this great place in order to strengthen the area of algorithm engineering.

The following collection of abstracts in chronological order gives a more detailed overview of the program.

Workshop: Algorithm Engineering

Table of Contents

Abstracts

Algorithm Engineering: A Personal and Historical Perspective

Giuseppe F. Italiano

In the last decades, we have witnessed an impressive progress in the field of computer algorithms. Many important and widely recognized achievements in several areas, including advanced data structures, computational geometry, cryptographic protocols, optimization algorithms, string searching and computational biology, were obtained. Despite this wealth of theoretical results, however, the transfer of algorithmic technologies has not experienced a comparable growth.

One reason for this may be that some of the algorithms designed are very hard to implement, and often suffer from large asymptotic constants: two characteristics which can make the effective deployment of algorithms somewhat problematic. Furthermore, measures of running times may be insufficient to design and analyze algorithms which are meant to be used in real applications. Toward this end, algorithm designers are starting to pay more attention to the details of the machine model that they use and to investigate new and more effective computational measures. Classical computational models, although effective, seem too idealized in many cases and are getting far apart from the recent developments in today's architectures. More detailed considerations seem to be needed to improve the actual running time of computations, such as external memory effects, caching, locality of data, pointer dereferencing, etc... etc...

As a result, more attention has been devoted to the engineering of algorithms, following an approach which has been largely referred to as *Algorithm Engineering.* Algorithm engineering consists of the design, analysis, experimental testing, tuning and characterization of robust algorithms: it is mainly concerned with issues of realistic algorithm performance, and studies algorithms and data structures by carefully combining traditional theoretical methods together with thorough experimental investigations.

There are many potential benefits involved in this approach. First of all, it promotes and fosters bridges toward key algorithmic applications. Furthermore, experimenting with algorithms and data structures has already proven to be a crucial step in many circumstances, such as in the case of heuristics for very hard combinatorial problems, design of test suites for a variety of problems, and for proposing new conjectures that may be of theoretical interest as well. Indeed, experimentation can provide guidelines to realistic algorithm performance whenever standard theoretical analyses fail. In our experience, experimentation is a very important step in the design and analysis of algorithms, as it tests many underlying assumptions and tends to bring algorithmic questions closer to the problems that originally motivated the work. Last, but not least, providing leading edge implementations of algorithms is also a key step for a successful technology transfer of algorithmic research.

Simulated Annealing versus Metropolis and the Black-Box Complexity of Search Problems

Ingo Wegener

Randomized search heuristics including simulated annealing, Metropolis algorithm, tabu search, and all variants of evolutionary and genetic algorithms find many applications in engineering and optimization, in particular, if the function to be optimized cannot be described in closed form. Algorithm engineering is necessary to find good values for their free parameters like the temperature of the Metropolis algorithm. For simulated annealing, a general convergence result is known leading only to exponential upper bounds for the expected optimization time even of quite simple problems. General tools to prove large lower bounds for certain instances are known. This fact leads Jerrum and Sinclair [4] to the following statement: "It remains an outstanding open problem to exhibit a natural example in which simulated annealing outperforms the Metropolis algorithm at a carefully fixed value of α." This open problem is solved by investigating the minimum spanning tree problem which is a natural problem of combinatorial optimization. In particular, it is proven that simulated annealing finds minimum spanning trees for a large class of instances with overwhelming probability. For many of these instances, the Metroplis algorithm needs exponential time with overwhelming probability.

Much monographs on evolutionary algorithms claim that these algorithms are particularly efficient in unimodal problems. This claim is disproved by showing that no randomized search heuristic can be efficient on all unimodal functions on $\{0,1\}^n$. A new brand of complexity theory, called black-box complexity, is developed for this purpose. The lower bound is based on Yao's minimax principle. The results have been published on ICALP '05 [1], FOGA '02 [2], and in Theory of Computing Systems [3].

References

[1] I. Wegener, *Simulated annealing beats Metropolis in combinatorial optimization*, ICALP 2005, LNCS **3580** (2005), 589–601.

[2] S. Droste, T. Jansen, K. Tinnefeld, and I. Wegener, *A new framework for the valuation of algorithms for black-box optimization*, FOGA 2002, Foundations of Genetic Algorithms **7** (2003), 253–270.

[3] S. Droste, T. Jansen, K. Tinnefeld, and I. Wegener, *Upper and lower bounds for randomized search heuristics in black-box optimization*, Theory of Computing Systems **4** (2006), 525–544.

[4] M. Jerrum and A. Sinclair, *The Markov chain Monte Carlo method: An approach to approximate counting and integration*, Chapter 12 of D. Hochbaum (Ed.) Approximation Algorithms for NP-hard Problems (1996), 482–522.

Engineering Succinct DOM

Rajeev Raman
(joint work with O'Neil Delpratt, Naila Rahman)

XML is a standard format for data exchange and storage. XML documents are processed by a number of applications in the following manner: the XML document is parsed, and a tree representation of the XML document is created within the memory of the computer. This representation is then accessed through the standard DOM (Document Object Model) interface, which allows a variety of navigational operations on the XML document. The DOM interface is very flexible, and is very commonly used for XML processing. Our focus is on *static* XML documents — while DOM does have functionality that allows (fairly arbitrary) changes to the XML document, this functionality not very frequently used. Indeed, there are a few DOM implementations for static documents.

A major disadvantage of most implementations of DOM is a high memory requirement, referred to as "XML bloat". The in-memory DOM representation of an XML document can be many times larger than the file storing the XML document (which in turn is a somewhat verbose representation of the underlying data and its relationships). This means that even moderately large XML documents cannot be processed within the main memory of a reasonably high-end machine. A primary cause for the "XML bloat" is that the DOM implementations use a largely pointer-based representation of the associations between the various data elements. For example, a single node in the tree representation of an XML document may have pointers to its parent, first- and last- children, and its next and previous siblings, among others.

We describe a DOM implementation that does not require the use of pointers, and is based upon *succinct* data structures. Succicnt data structures use the information-theoretically minimum number of bits to encode a object. For example, an *ordinal* tree on n nodes is a rooted tree, where the children of a node are ordered from left-to-right (XML documents are essentially ordinal trees). Since there are $t_n = \frac{1}{n}\binom{2n-2}{n-1}$ n-node ordinal trees, it follows that an ordinal tree on n nodes must be represented in $\lg t_n = 2n - O(\log n)$ bits ($\lg x = \log_2 x$). On the other hand, representing the tree structure of an XML document using pointers (as described above) would use asymptotically $5n \lg n$ bits; from a practical perspective, each pointer would require either 32 or 64 bits, so one would require $160n$ or $320n$ bits for the tree structure for practical values of n.

We use a succinct tree representation developed in [2, 1]. We also use succinct data structures in place of explicit pointers to replace pointers to lists of attributes [3], and to textual data associated with text nodes. However, the final data structure turned out not to be a simple assembly of these components. A number of XML-motivated optimizations were included, including fast tests for leaf nodes, so-called *double-numbering* of nodes, consideration of the distributions of the lengths of textual data in text nodes, attributes and comment nodes, and fast "shortcuts" for node iterators.

We tested a preliminary version of succinct DOM by reading and storing an XML document in main memory, and traversing it. Excluding textual data from consideration, the size of the XML document is reduced by usually a factor of 10 relative to the file size (and is even smaller relative to the in-memory representation size), while traversals are only about 3-4 times slower, relative to a pointer-based XML representation.

References

[1] O. Delpratt, N., and R. Raman, *Engineering the LOUDS Succinct Tree Representation*, Proc. WEA '06, LNCS **4007** (2006), 134–145.
[2] R.F. Geary, N. Rahman, R. Raman, and V. Raman, A simple optimal representation for balanced parentheses, Theor. Comput. Sci. **368**(3), (2006), 231–246.
[3] Compressed Prefix Sums, O. Delpratt, N. Rahman, and R. Raman, *Compressed Prefix Sums*, Proc. SOFSEM 2007, LNCS **4362** (2007), 235–247.

Algorithm Engineering — An Attempt at a Definition

Peter Sanders
(joint work with Kurt Mehlhorn, Rolf Möhring, Burkhardt Monien, Petra Mutzel, and Dorothea Wagner)

Algorithm Engineering (AE) is a methodology for algorithmics (the subdiscipline of computer science devoted to the development of efficient algorithms) summarized in Figure 1. The core of AE is a cycle consisting of design, analysis, implementation and experimental evaluation of algorithms. In addition, we need realistic models, reusable algorithm libraries and all this allows a closer coupling to applications. Implementations can be used in applications — directly or via algorithm libraries. In turn, applications contribute real world problem instances and motivations for realistic models.

The term AE was first used in the end of the 1990s (e.g. call for papers of WAE 1997) in the algorithm theory community. The background is, that beginning in the early 1990s, it was observed that there was a growing gap between algorithm theory and the algorithms actually used in practice. (Complicated algorithms for simple machine and problem models designed for good asymptotic worst case performance versus simple algorithms for real world machines and problems whose performance evaluation was mostly experimental and where constant factors matter.) As a consequence, it was more and more frequently demanded that algorithmics should include also implementation of algorithms and experimental evaluation using real world problem instances. AE is sometimes viewed as a synonym for *experimental algorithmics*. However, even in the earliest references, it is stressed that experiments can influence models and design and that also algorithm analysis might look at different questions like average case performance, families of easy problem instances, or constant factors in the execution time. Therefore, the wider view proposed here that views AE as a methodology containing all of algorithmics, seems to be a more apt definition.

There are regular scientific conferences dedicated to algorithm engineering: WEA (Workshop on Algorithm Engineering, now ESA applied track) since 1997, Alenex (Algorithm Engineering and Experimentation) since 1999 and WEA (Workshop on Experimental Algorithms) since 2001. A longer version of this text (in German) can be found at the home page of the DFG focus project on algorithm engineering `www.algorithm-engineering.de`.

Engineering Route Planning Algorithms

Dominik Schultes
(joint work with Peter Sanders)

The computation of shortest paths in a graph is a well-known problem in graph theory. One of the most obvious practical applications is route planning in a road network, i.e., finding an optimal route from a start location to a target location. It is often assumed that a given road network does not change very often and that there are many source-target queries on the same network so that it pays to invest some time for a preprocessing step that accelerates all further queries. Based on this assumption, we developed various speedup techniques for route planning.

Highway hierarchies [1, 2, 3] exploit the hierarchy inherent in real-world road networks. In a preprocessing step, we investigate the given road network in order to extract and prepare a hierarchical representation. Our route planning algorithm then takes advantage of this data. It is an adaptation of the bidirectional version of Dijkstra's algorithm, massively restricting its search space.

In several experiments, we concentrate on the computation of fastest routes in Western Europe and the USA. Both networks consist of about 20 million nodes

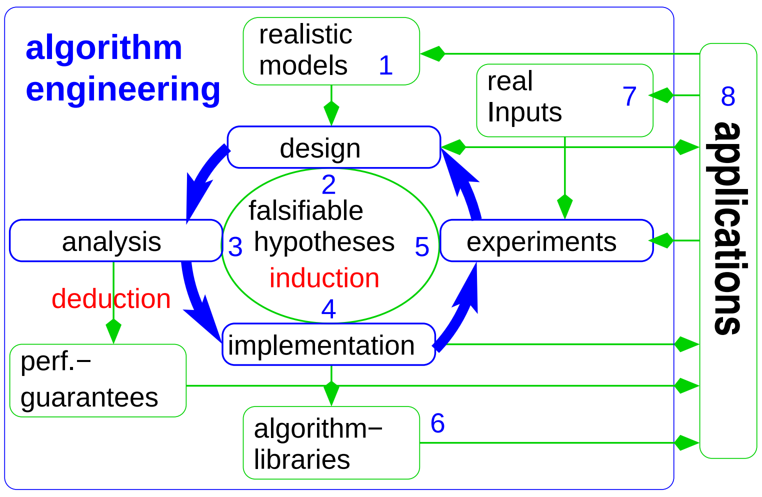

Figure 1.

each. Our algorithm preprocesses these networks in 20 minutes using linear space. Queries then take less than one millisecond to produce optimal routes. This is more than 7 000 times faster than using Dijkstra's algorithm.

A combination [4] with a goal-directed approach, namely landmark-based A*-search, yields a slight reduction of the query times. In particular, such a combination is useful when we deal with approximate queries or with a distance metric (instead of the usual travel time metric).

A many-to-many variant [5] of the highway hierarchies is capable of computing distance tables that contain for given source and target node sets the shortest path distances between all source-target pairs. For example, a $10\,000 \times 10\,000$ table can be filled in about one minute.

Transit node routing [6, 7, 8] is based on the following observation: "When you drive to somewhere 'far away', you will leave your current location via one of only a few 'important' traffic junctions [*transit nodes*]". Distances from each node to all neighbouring transit nodes and between all transit nodes are precomputed so that a non-local shortest-path query can be reduced to a small number of table lookups. That way, average query times can be reduced to around five microseconds, which is about one million times faster than using Dijkstra's algorithm.

Highway-node routing [9] is a *dynamic* technique for fast route planning in large road networks. For the first time, it is possible to handle the practically relevant scenarios that arise in present-day navigation systems: When an edge weight changes (e.g., due to a traffic jam), we can update the preprocessed information in 2–40 ms allowing subsequent fast queries in about one millisecond on average. When we want to perform only a single query, we can skip the comparatively expensive update step and directly perform a prudent query that automatically takes the changed situation into account. If the overall cost function changes (e.g., due to a different vehicle type), recomputing the preprocessed information takes typically less than two minutes.

The foundation of our dynamic method is a new static approach that generalises and combines several previous speedup techniques. It has outstandingly low memory requirements of only a few bytes per node.

References

[1] D. Schultes, *Fast and exact shortest path queries using highway hierarchies*, Master's thesis, Universität des Saarlandes, 2005.

[2] P. Sanders and D. Schultes, *Highway hierarchies hasten exact shortest path queries*, 13th European Symposium on Algorithms, LNCS **3669** (2005), 568–579.

[3] P. Sanders and D. Schultes, *Engineering highway hierarchies*, 14th European Symposium on Algorithms, LNCS **4168** (2006), 804–816.

[4] D. Delling, P. Sanders, D. Schultes, and D. Wagner, *Highway hierarchies star*, 9th DIMACS Implementation Challenge [10], 2006.

[5] S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner, *Computing many-to-many shortest paths using highway hierarchies*, Workshop on Algorithm Engineering and Experiments, 2007.

[6] P. Sanders and D. Schultes, *Robust, almost constant time shortest-path queries in road networks*, 9th DIMACS Implementation Challenge [10], 2006.

[7] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes, *In transit to constant time shortest-path queries in road networks*, Workshop on Algorithm Engineering and Experiments, 2007.

[8] H. Bast, S. Funke, P. Sanders, and D. Schultes, *Fast routing in road networks with transit nodes*, Science **316**(5824) (2007), 566.

[9] D. Schultes and P. Sanders, *Dynamic highway-node routing*, 6th Workshop on Experimental Algorithms, 2007.

[10] 9th DIMACS Implementation Challenge, *Shortest Paths*, `http://www.dis.uniroma1.it/~challenge9/`, 2006.

An ILP Formulation for 2-Root-Connected Prize-Collecting Steiner Networks using Directed Cuts

Markus Chimani
(joint work with Maria Kandyba and Petra Mutzel)

Extending already existing fiber-optics networks by connecting new customers is an important topic in the design of telecommunication networks. Thereby, we have an existing infrastructure network I, a set of potential new customers C and a set of potential new route-segments for laying the fiber cables. As each new customer v will generate a certain assessable profit $p(v) \in \mathbb{R}^+$ and each route-segment e has a certain laying cost $c(v) \in \mathbb{R}^+$, the main task is to connect a subset of C with I such that the overall profit is maximized. In this paper we consider the real world problem, where some of the customers, if added to the network, require two node-disjoint connections to I to increase reliability. We denote these customers with the set C_2, and the other customers with C_1.

By representing the infrastructure network by a single root node r, we obtain a rooted Prize-Collecting Steiner Network problem where certain nodes are required to be (nodewise) 2-connected with the root. Formally, we are given an undirected graph $G = (V, E)$, a root node $r \in V$, a set of customer nodes $C = C_1 \dot{\cup} C_2 \subset V$, a prize function $p : V \to \mathbb{R}^+$, and a cost function $c : E \to \mathbb{R}^+$. Find a subgraph $N = (V_N, E_N)$ of G with $r \in V_N$ which minimizes $\sum_{e \in E_N} c(e) - \sum_{v \in V_N} p(v)$ and satisfies the following connectivity property: for every node $v \in C_k \cap V_N$ ($k \in \{1, 2\}$), N contains at least k node-disjoint paths connecting v to r.

We call such a problem a *2-Root-connected Prize-Collecting Steiner Network* problem (2RPCSN). If we require all customers to be included into the solution network, the resulting problem is called *2-Root-connected Steiner Network* problem (2RSN). Both 2RPCSN and 2RSN are NP-hard, as they contain the Steiner tree problem as a special case. While we concentrate on the investigation of 2RPCSN, all results clearly also hold for 2RSN. Furthermore, our approach can be used for the relaxed version where C_2 customers are only required to be 2-edge-connected with the root.

2RPCSN was already studied by Wagner et al. and two different ILP formulations for this problem were suggested: one based on multi-commodity flow, the other one using undirected cut inequalities. We report on a transformation of 2RPCSN into the problem of finding an optimal subgraph in a related directed graph and give a new ILP formulation which uses directed cut inequalities, see [1] for details. To our knowledge, our formulation is the first which applies such an approach to a node-disjoint connectivity problem. The central idea is based on proving that for each 2-node-connected graph G with a given root r, there exists an orientation of G such that each node lies on a simple directed cycle with r.

Furthermore, we study the polyhedral properties of our ILP and show that our formulation is stronger than the undirected cut formulation. We solve 2RPCSN using this new formulation within a Branch-and-Cut framework, utilizing an LP-based heuristic also presented herein. Our experimental results show that our approach is superior to those of Wagner et al. for nearly all test instances.

Finally, we briefly describe how this approach for 2RPCSN can be used to give the first directed formulation for the 2NCON problem, i.e., every customer has to become connected, and all C_2 customers have to become 2-node-connected with each other: therefore we can select any C_2 node as the root, and compute a network with the property that this root has only a single incoming edge. We also sketch how to use this approach to solve even the 2PCSN problem, i.e., the 2NCON problem with the prize-collecting property for each customer.

References

[1] M. Chimani, M. Kandyba, P. Mutzel, *A New ILP Formulation for 2-Root-Connected Prize-Collecting Steiner Networks*, Technical Report TR07-1-001, University Dortmund, Chair for Algorithm Engineering, 2007.

Software Engineering for Mathematical Software?

Thorsten Koch

1. Attitudes

Algorithm engineering refers to the process required to transform a pencil-and-paper algorithm into a robust, efficient, ***well tested****, and easily usable implementation.*

– Bader, Moret, Sanders [1]

How much software engineering, especially testing and code tuning is needed in (mathematical) research software? Several books on testing and software engineering are among the longest selling books in computer science. *The Art of Software testing* [2] has been updated to a second edition after being in print for more than 25 years. *The mythical man month* [3] is available in a 20 year anniversary edition since 1995. According to the Wikipedia this book is called the bible of software engineering, since *"everybody reads it, but nobody does anything about it"*. The attitudes towards testing code can be summarized as follows:

Real Programmers don't comment their code. If it was hard to write, it should be hard to understand and harder to modify.

– `fortune(6)`

Beware of bugs in the above program. I have only proved it correct, not tried it.

– D.E.Knuth

The single most important rule of testing is to do it.

– Kernighan, Pike [4]

How much testing is enough for software developed in research? Empirically, for research software a program is tested enough if it gives plausible results on all instances the author regards as indispensable. This might be deemed acceptable as long as nobody else uses the program. Since advances in research depend on the ability to build upon the work of others, it is, however, highly favorable to share the software developed. Unfortunately, this will require you to test. What are the possibilities?

- ▶ Use the classical "banana approach" from the software industry:
  Product matures at the customer.
- ▶ Use the open source approach:
  Hope users will find bugs and send patches.
- ▶ **Test yourself!**

2. Testing

We use the program Zimpl[1] [5] has an example. Zimpl has about 10,000 lines of code. Asserts are used to state pre- and post-conditions and some invariants. There is one Assert every 6 statements, on average. Currently, more than 150 tests are used to exercise the code. Special care has been taken to write a test for every error message the program can give. This process led to interesting results as several bugs were found and error messages were discovered that could not possibly be produced. Currently, the total test coverage[2] is 86%. Hence, about 1,400 lines of code are never executed by any of the tests. These are usually parts of the code that handle special situations as, for example, buffer reallocation for very long input lines. A program tested with normal input typically has a coverage of about 50%. With a wide range of input data up to 70% can often be achieved. Beyond this, options and error conditions have to be systematically exercised. Tedious work, indeed! Adding one function means to add at least one functional test, often there are one or two error conditions, and finally it has to be documented. Popular estimates state that producing well-tested and documented code requires about three times as much effort than just "the plain code". The same factor is usually attributed to making code reusable. This means if you want to produce well-tested, documented, and reusable code, then the work needed will be about one order of magnitude bigger than just writing a program that does the job (at

[1]`http://zimpl.zib.de`

[2]See the GCOV command at `http://gcc.gnu.org`

least sometimes). Nevertheless, if this is the goal, (automated) regression test are extremely useful. This is true in particular for software that is intended to be developed further. Regression test that have a high coverage in combination with the use of automated software testing (like valgrind) can considerable improve the quality of software.

Note, though, that coverage tests say nothing about correctness of the program. Also regression tests only verify that the code behaves the same than before any changes, regardless whether this behavior was correct or not. In any case functional tests are needed. This is relatively easy for a program such as Zimpl, but can be very difficult for programs that compute something new. In an ongoing project at ZIB the goal is to compute all solutions to a binary integer program. Here, the design of particular tests is necessary as there are only few programs available to verify the results computed.

3. Tuning

Assuming the theoretically best algorithm is known: How should it be implemented?

- Which language? Does this matter?
- Use libraries or implement yourself? Especially system libraries are very sensitive in the sense that their performance might vary considerably between compilers and operating systems. The `push_back` operator of the STL can behave very differently between compilers. `malloc(3)` is well known to be a performance hazard when porting software.
- Implement special cache/hardware aware algorithms?
- Use assembler, SSE/2/3, 3DNow, Altivec, etc.? Assembler intrinsics can, for example, speed up operations on bits considerably if there is an machine instruction available for the operation. Finding the leftmost bit in a word can be expressed by a single instruction on many processors, while this is often cumbersome to program in a higher language.
- Parallelize it? How? By hand, OpenMP, MPI, ...
  Architectures that support multi-threading are common today and will spread further in the future. Making aware of this wealth of computing power is a major topic. Still, the problems also increase: testing and debugging parallel software is a nightmare, reproducible time measurements on parallel systems with non-uniform memory access (NUMA) are nearly impossible to achieve.
- Single Instruction Multiple Data (SIMD) processors, like the 8800 GPU, the Clearspeed accelerator or the Cell processor, promise ten times the performance of regular processors, provided the problem is suitable and can be implemented on these rather restricted architectures. Assume a mediocre implementation of an algorithm would take 1 minute to run. A better implementation might need only 30 seconds. Using some processor specific instructions and some tuning, it would be done in 10 s. A multi-threaded implementation would run in 3 s, and using SIMD hardware it

would work in half a second. Whether the computation needs a minute or can be done twice a second often makes a big difference in the possible applications of the algorithm.

- And last but not least, the performance of algorithms depends on the input data. Analyzing typical instances and test runs can give considerable ideas for speedup. How data dependent should the program get?

References

[1] D.A. Bader, B.M.E. Moret, and P. Sanders, *Algorithm Engineering for Parallel Computation*, in R. Fleischer, B. Moret, E. Meineche Schmidt (Eds.): *Experimental Algorithmics, From Algorithm Design to Robust and Efficient Software*, LNCS **2547** (2002), 1–23.

[2] G.J. Myers, C. Sandler, T. Badgett, and T.M. Thomas, *The Art of Software Testing*, 2nd Ed., Wiley, 2004.

[3] F.P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, 20th Anniversary Ed., Addison-Wesley, 1995.

[4] B.W. Kernighan, R. Pike, *The Practice of Programming*, Addison-Wesley, 1999

[5] T. Koch, *Rapid Mathematical Programming* PhD thesis, Technische Universität Berlin, 2004, `http://www.zib.de/Publications/abstracts/ZR-04-58`

The TSP and Exact Computation

William Cook
(joint work with David Applegate, Robert Bixby, Vasek Chvátal, Sanjeeb Dash, Daniel Espinoza, Ricardo Fukasawa, Marcos Goycoolea, and Keld Helsgaun)

The traveling salesman problem asks for the cheapest tour passing through each of a finite set of cities and returning to the point of departure. We give a brief survey of the history and applications of the TSP, including work on genome sequencing, and report on the solution of the full set of TSPLIB challenge problems, the largest instance having 85,900 cities arising in a VLSI application. We also discuss the solution of geometric TSP instances with exact (real) Euclidean travel costs and make an estimation of the Beardwood, Halton, and Hammersley TSP constant. To treat very large instances of the TSP we describe decomposition techniques for computing tight upper and lower bounds on optimal tour values. Finally, we consider the use of Gomory mixed-integer cutting planes for improving TSP relaxations.

References

[1] D. Applegate, R. Bixby, V. Chvátal, W. Cook, *The Traveling Salesman Problem: A Computational Study*, Princeton University Press, Princeton, New Jersey, USA, 2006.

[2] D. Applegate, W. Cook, S. Dash, D. Espinoza, *Exact solutions to linear programming problems*, Operations Research Letters (2007), to appear.

[3] J. Beardwood, J. H. Halton, J. M. Hammersley, *The shortest path through many points*, Proceedings of the Cambridge Philosophical Society **55** (1959), 299–327.

[4] W. Cook, D. Espinoza, M. Goycoolea, *Computing with domino-parity inequalities for the TSP*, INFORMS Journal on Computing, accepted in 2006.

[5] K. Helsgaun, *An effective implementation of the Lin-Kernighan traveling salesman heuristic*, European Journal of Operational Research **126** (2000), 106–130.

[6] G. Reinelt, *TSPLIB—A traveling salesman problem library*, ORSA Journal on Computing **3** (1991), 376–384.
[7] A. Schaeffer, E. Rice, W. Cook, R. Agarwala, *rh_tsp_map 3.0: End-to-end radiation hybrid mapping with improved speed and quality control*, Bioinformatics (2007), to appear.

An Experimental Study of New and Known Online Packet Buffering Algorithms

Susanne Albers
(joint work with Tobias Jacobs)

Over the past five years the algorithms community has witnessed tremendous research interest in packet buffering algorithms. Given a network router or switch that is equipped with packet buffers of limited capacity, the general goal is to design strategies for serving these buffers so as to maximize the total packet throughput. While packet buffering policies have been investigated in the applied computer science and, in particular, networking communities for many years, only seminal papers by Aiello et al. [1] and Kesselman et al. [5] have initiated theoretical and algorithmic studies. These studies aim at analyzing existing algorithms and at designing new strategies with a provably good performance.

Obviously, packet buffering is an online problem in that data packets arrive over time and, at any time, future packet arrivals are unknown. Results from queueing theory cannot be applied as network traffic exhibits so-called *self-similar* properties. Therefore, one resorts to competitive analysis [7], comparing an online algorithm A to an optimal offline algorithm OPT that knows the entire packet arrival sequence in advance. Algorithm A is called c-competitive if, for all packet arrival sequences, the throughput achieved by A is at least $1/c$ times that of OPT. In the above-mentioned algorithmic body of work, various packet buffering problems were investigated. The following natural questions arise: Do the competitive analyses give meaningful results? Are the proposed new algorithms interesting from a practical point of view? Does optimizing the worst-case behaviour also improve the practical performance? So far, these issues were not addressed.

In this paper we present the first experimental study of online packet buffering algorithms. We consider a scenario that is very basic and has been investigated the most among the proposed models. Specifically, we are given m packet buffers, each of which is associated with an input port of a switch. Each buffer is organized as a queue and can simultaneously store up to B data packets. The capacity B is also referred to as the *size* of the buffer. Time is assumed to be discrete. Each time step consists of two phases, namely a *packet arrival phase* and a *packet transmission phase*. At any time, in the packet arrival phase, new packets may arrive at the buffers. Let b_i be the number of packets currently stored in buffer i, and let a_i be the number of newly arriving packets at that buffer. If $a_i + b_i \leq B$, then all new packets can be accepted; otherwise $a_i + b_i - B$ packets must be dropped. Furthermore, at any time, in the packet transmission phase, an algorithm can select one non-empty buffer and transfer the packet at the head of that queue to the output port. We assume w.l.o.g. that the packet arrival phase precedes the transmission phase. The goal is to maximize the throughput, i.e. the total number of transferred packets.

The scenario we study here arises, for instance, in input-queued (IQ) switches which represent the dominant switch architecture today. In an IQ switch with m input and m output ports packets that arrive at input i and have to be routed to output j are buffered in a virtual output queue Q_{ij}. In each time step, for any output j, one data packet from queues Q_{ij}, $1 \leq i \leq m$, can be sent to that output. In our problem formulation the m buffers correspond to queues Q_{ij}, $1 \leq i \leq m$, for any fixed j. We emphasize that we consider all packets to be equally important, i.e. all of them have the same value. Most current networks, in particular IP networks, treat packets from different data streams equally in intermediate switches.

Known algorithms: The most simple and natural packet buffering algorithm is the *Greedy* policy: At any time serve the queue currently storing the largest number of packets. Unfortunately, *Greedy* has essentially the worst possible competitive ratio. It is easy to show that any *work conserving* algorithm, which at any time serves an arbitrary non-empty buffer, is 2-competitive. Obviously, *Greedy* belongs to the class of work conserving strategies. It was shown in [4] that the competitive ratio of *Greedy* is not smaller than $2-1/B$, no matter how ties are broken. Thus *Greedy* has a competitiveness of exactly 2, for arbitrary buffer sizes. The first deterministic algorithm that achieved a competitive ratio below 2 was devised in [4]. The proposed *Semi Greedy* algorithm deviates from standard *Greedy* when the buffer occupancy is low and has a competitive performance of $17/9 \approx 1.89$. The deterministic strategy with the smallest competitive ratio known is the *Waterlevel* algorithm [2] with a competitiveness of $\frac{e}{e-1}(1+\frac{\lfloor H_m+1 \rfloor}{B})$, where H_m is the m-th Harmonic number. This ratio is optimal, for large B, as no deterministic algorithm can have a competitive ratio smaller than $e/(e-1) \approx 1.58$, see [4]. As for randomized strategies, a *Random Schedule* algorithm [3] achieves a competitive ratio of $e/(e-1)$ while a *Random Permutation* algorithm is 1.5-competitive [6]. These performance ratios hold against oblivious adversaries and are close to the best lower bound of 1.46, see [4]. The five algorithms just mentioned comprise all online strategies known in the literature for our packet buffering problem. As for the offline problem, a polynomial time algorithm computing optimal solutions was given in [4].

Our contributions: We first introduce a new online packet buffering algorithm called *HSFOD*. It is based on the idea to estimate the packet arrival rate for each port. In each time step the algorithm transmits a packet from a non-empty queue that, according to these arrival rates, encounters packet loss earliest in the future assuming buffers would not be served anymore. We prove that it achieves a competitive ratio of 2.

The major part of this paper is devoted to an extensive experimental study of the packet buffering problem under consideration. The main purpose of our experiments is to determine the experimentally observed competitiveness of all the proposed online algorithms and to establish a relative performance ranking among the strategies. As the name suggests, the experimentally observed competitiveness is the ratio of the throughput of an online algorithm to that of an optimal solution as it shows in experimental tests. Additionally, we wish to evaluate the running times and memory requirements of the algorithms as some of the strategies are quite involved and need auxiliary data structures. Finally, we are interested in the actual throughput in terms of the total number of successfully transmitted packets.

In order to get realistic and meaningful results, we have tested the algorithms on real-world traces. We selected traces from the Internet Traffic Archive, which is a moderated trace repository sponsored by ACM SIGCOMM. In our experiments we have studied varying port numbers m as well as varying buffers sizes B. Furthermore, we have investigated the influence of varying the *speed* of a switch, i.e. the frequency with which it can forward packets. We have adjusted this parameter relative to the given data traces. For instance, a speed of value 1 indicates that the *average* packet arrival frequency is equal to the frequency with which packets can be transmitted.

We present a concise description of the five previously known online buffering algorithms as well as the optimal offline strategy. For all the proposed strategies, including *HSFOD*, we describe how the given pseudo-code was indeed implemented and discuss runtime issues as well as extra space requirements of the strategies. We implemented the data model and the algorithms using the Java programming language. One of the most important findings is that the experimentally observed competitiveness is much lower than the theoretical bounds. Typically, the online algorithms are at most 3% worse than an optimal offline algorithm. In fact, *HSFOD* shows the best performance, having a gap of less and 0.1%. We remark here that *HSFOD* was designed after we had implemented and evaluated the previously known algorithms. Hence it can be viewed as a result of an algorithm engineering process. Furthermore, the theoretical competitive ratios are no proper indication of how the algorithms perform in practice. The randomized algorithms, despite their low theoretical competitiveness, do not perform better than the deterministic ones. From a practical point of view *Greedy*, *HSFOD* and *Semi Greedy* are the algorithms of choice.

References

[1] W. Aiello, Y. Mansour, S. Rajagopolan and A. Rosén, *Competitive queue policies for differentiated services*, Proc. INFOCOM (200), 431–440.

[2] Y. Azar and A. Litichevskey, *Maximizing throughput in multi-queue switches*, Proc. 12th Annual European Symp. on Algorithms (ESA), LNCS **3221** (2004), 53–64.

[3] Y. Azar and Y. Richter, *Management of multi-queue switches in QoS networks*, Proc. 35th ACM Symp. on Theory of Computing, 2003, 82–89.

[4] S. Albers and M. Schmidt, *On the performance of greedy algorithms in packet buffering*, Proc. 36th ACM Symp. on Theory of Computing, 2004, 35–44.

[5] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber and M. Sviridenko, *Buffer overflow management in QoS switches*, Proc. 31st ACM Symp. on Theory of Computing, 2001, 520–529.

[6] M. Schmidt. Packet buffering, *Randomization beats deterministic algorithms*, Proc. 22nd Annual Symp. on Theoretical Aspects of Computer Science (STACS), LNCS **3404** (2005), 293–304.

[7] D.D. Sleator and R.E. Tarjan, *Amortized efficiency of list update and paging rules*, Comm. of the ACM **28** (2005), 202–208.

Constrained Minkowski Sums

Friedrich Eisenbrand

(joint work with Thorsten Bernholt and Thomas Hofmeister)

The *Minkowski sum* of two (finite) point-sets $P \subseteq \mathbb{R}^2$ and $Q \subseteq \mathbb{R}^2$ is defined as $P \oplus Q = \{p+q \mid p \in P, q \in Q\}$. Convex hulls of Minkowski sums are a fundamental concept in algorithmic geometry, in particular in robot motion planning [5, 4, 6, 7] and placement problems [1, 3]. The convex hull of $P \oplus Q$ can be computed in linear time [5] if the points in P and Q are sorted w.r.t. the value of a given linear function, for example the value of their x_1-coordinate. The convex hull of $P \oplus Q$ has at most $N = |P| + |Q|$ vertices.

In this paper, we introduce the notion of a *constrained* Minkowski sum. For a matrix $A \in \mathbb{R}^{k \times 2}$ and a vector $b \in \mathbb{R}^k$, the *constrained Minkowski sum* $(P \oplus Q)_{A\,x \geq b}$ is defined as the point-set

$$(P \oplus Q)_{A\,x \geq b} = \{x \in P \oplus Q \mid A\,x \geq b\}.$$

If $k = 1$, then the system $A\,x \geq b$ reduces to a linear inequality $a^T x \geq \beta$. We call a constraint $a^T x \geq \beta$ *linearly sortable* if each $|a^T p|$, $p \in P$ and $|a^T q|$, $q \in Q$ is an integer bounded by $O(N)$.

Our motivation to study constrained Minkowski sums comes from a very practical application. A large class of subsequence problems from computational biology can be solved by maximizing a quasiconvex function over the points in a constrained Minkowski sum. Recall that a function $f : D \to \mathbb{R}$ is called *quasiconvex* if for all points $s_1, s_2 \in D$ and all $\lambda \in [0,1]$, one has $f(\lambda \cdot s_1 + (1-\lambda) \cdot s_2) \leq \max\{f(s_1), f(s_2)\}$, where $D \subseteq \mathbb{R}^2$ is a nonempty convex set. If $R \subseteq \mathbb{R}^2$ is a finite set of points, then f attains its maximum on one of the vertices of the *convex hull* $\mathrm{conv}(R)$ of R.

Our main results are as follows:

i) We show that the convex hull of a constrained Minkowski sum which is defined by one constraint can be computed in time $O(N \log N)$. If the constraint $a^T x \geq \beta$ is linearly sortable, our algorithm even achieves a running time of $O(N)$.

ii) We provide a tight upper bound on the number of vertices of the convex hull of a Minkowski sum with one constraint.

iii) We show that a subset R of $(P \oplus Q)_{Ax \geq b}$ which contains all vertices of $\mathrm{conv}\left((P \oplus Q)_{Ax \geq b}\right)$ can be computed in time $O(N \log N)$ if the number of constraints is fixed. This shows that a *quasiconvex* function which can be evaluated in constant time can be maximized over $(P \oplus Q)_{Ax \geq b}$ in time

$O(N \log N)$. This running time is best possible in the algebraic decision-tree model. For a varying number k of constraints we can compute such a set R in time $O(k \cdot \log k + k \cdot N \log N)$. The set R contains $O(k \cdot N \log N)$ points.

iv) We obtain for many subsequence problems from the literature linear time algorithms and improve upon the best known running times for some of them. The strength of our approach is that we have *one* algorithm which can be used for *all* of those problems.

Our result iii) for a fixed number of constraints is best possible in the *algebraic decision-tree* model. Ben-Or [2] showed that the *set-disjointness* problem has a lower bound of $\Omega(n \log n)$ in this model of computation. Set disjointness is defined as follows. Given two sets $A = \{a_1, \dots, a_n\} \subseteq \mathbb{R}$ and $B = \{b_1, \dots, b_n\} \subseteq \mathbb{R}$, one has to decide whether $A \cap B = \emptyset$ holds. Set-disjointness can be reduced to the problem of maximizing a quasiconvex, even linear, function over a constrained Minkowski sum in linear time as follows. Construct the point-sets $P = \{(0, -a) \mid a \in A\}$ and $Q = \{(0, b) \mid b \in B\}$. The point $(0, 0)$ is contained in $P \oplus Q$ if and only if A and B are not disjoint. Thus the maximum of the objective function $-x_2$ over the constrained Minkowski sum $(P \oplus Q)_{x_2 \geq 0}$ is equal to 0 if and only if A and B are not disjoint. This shows that the problem of maximizing a quasiconvex objective function over the constrained Minkowski sum $(P \oplus Q)_{Ax \geq b}$ requires time $\Omega(n \log n)$ in the algebraic decision-tree model even if f is a linear function and $Ax \geq b$ consists of only one constraint.

References

[1] P. K. Agarwal, N. Amenta, and M. Sharir, *Largest placement of one convex polygon inside another*, Discrete Comput. Geom. **19**(1) (1998), 95–104. MR MR1486639 (99c:52026)

[2] M. Ben-Or, *Lower bounds for algebraic computation trees*, Proceedings of the 15th Annual ACM Symposium on Theory of Computing, STOC'83 (Boston, MA, May 25-27, 1983) (New York), ACM, ACM Press, 1983, 80–86.

[3] L. Paul Chew and Klara Kedem, *A convex polygon among polygonal obstacles: placement and high-clearance motion*, Comput. Geom. **3**(2) (1993), 59–89. MR MR1228772 (94i:68283)

[4] Leonidas J. Guibas, Micha Sharir, and Shmuel Sifrony, *On the general motion-planning problem with two degrees of freedom*, Discrete Comput. Geom. **4**(5) (1989), 491–521. MR MR1014740 (91a:68268)

[5] L.J. Guibas, L.H. Ramshaw, and J. Stolfi, *A kinetic framework for computational geometry*, Proceedings of 24th IEEE Symposium on the Foundations of Computer Science, 1983, 100–111.

[6] J. C. Latombe, *Robot motion planning*, Kluver Academic Publishers, Boston, MA, 1991.

[7] T. Lozano-Perez and M. A. Wesley, *An algorithm for planning collision-free paths among polyhedral obstacles*, Communications of the ACM **22** (1979), 560–570.

The Core Concept and Collaborative Approaches for the Multidimensional Knapsack Problem

Ulrich Pferschy
(joint work with Jakob Puchinger and Günther R. Raidl)

We study the multidimensional knapsack problem (MKP) where a subset of items with maximum profit has to be selected from a given ground set subject to m resource constraints. The strongly $\mathcal{N}P$-hard MKP can be defined by the following integer linear program:

$$\text{(1)} \qquad \text{(MKP)} \qquad \text{maximize} \quad \sum_{j=1}^{n} p_j x_j$$

$$\text{(2)} \qquad \text{subject to} \quad \sum_{j=1}^{n} w_{ij} x_j \le c_i, \quad i = 1, \dots, m,$$

$$\text{(3)} \qquad x_j \in \{0,1\}, \quad j = 1, \dots, n.$$

Each item j consumes an amount $w_{ij} \ge 0$ from each resource i, which is bounded by the capacity c_i. All input values are assumed to be nonnegative integers. Obviously, for $m = 1$ the classical knapsack problem (KP) arises as a special case. Recent surveys on MKP can be found in Kellerer, Pferschy and Pisinger [3] and Fréville [2].

At first we study the structure of the LP-relaxation where (3) is replaced by $x_j \in [0,1]$. It follows from linear programming theory that the solution of the LP-relaxation x^{LP} has at most m fractional values. An empirical study on classical benchmark instances (all instances can be found in Beasley's OR-library cf. [1]) shows that the integral part of x^{LP} and the optimal solution of (MKP) denoted by x^{ILP} coincide for roughly 97% of the variables. On the other hand, extending x^{LP} by a greedy-type heuristic yields solutions with a Hamming distance of 10% from x^{ILP} on average. Therefore, we try to guide the branch and bound approach performed by CPLEX 9.0 to concentrate on a neighbourhood of x^{LP} before exploring other parts of the solution space. This so-called local branching approach yields consistently better solutions than running CPLEX in default mode. It turned out that the neighbourhood should be neither too small nor too large with a bound of 25 on the Hamming distance from x^{LP} exhibiting the best performance.

For the one-dimensional knapsack problem (KP) so-called core algorithms are the currently most successful solution procedures. They are based on the idea that items with a very high efficiency $e_j = p_j/w_j$ will always be included in the optimal solution whereas items with a low efficiency will never be. The remaining items are usually grouped around the split item which is defined by the only fractional value of the LP-relaxation. This set of items, where x^{LP} and x^{ILP} differ, are defined as the core of the problem. Since x^{ILP} is not known a priori an approximate core has to be defined by selecting a reasonably large set of items with efficiencies near the split item and solve the resulting smaller instance of (KP) where all items outside the approximate core are fixed.

An analogous approach for MKP has not been pursued in the literature before. A major point of consideration is the required definition of an efficiency measure to partition the items into highly desirable, possible and unlikely candidates for the optimal solution. In an experimental study we compare several efficiency measures suggested in the literature (see [3]). They are all based on different choices of relevance values r_i to weigh the importance of the i-th constraint and can be written as

$$e_j^{\text{general}} = \frac{p_j}{\sum_{i=1}^{m} r_i w_{ij}} \,. \tag{4}$$

Our computational results clearly indicate that setting r_i equal to the optimal dual solution value of the LP-relaxation yields the best results in the sense that the core size is much smaller than for all other efficiency values which generate cores almost twice as large.

To construct an approximate core for KP the split item is a natural center to start from. For MKP we define a split interval which is bounded (after sorting by efficiencies) by the fractional variables in x^{LP} with lowest resp. highest index. With complementary slackness it can be shown that using the dual solution as relevance values generates the theoretically smallest possible split interval. Our computational experiments also indicate that the center of the split interval is very close to the center of the exact core (less than 3 items difference in average) and that other efficiency values lead to far larger split intervals.

Therefore, we construct approximate cores symmetric around the center of the split interval. Solving cores of different size to optimality shows that a core containing 40% of the items yields almost always globally optimal solution but consumes in average only 40% of the running time required for the solution of the complete instances. Smaller cores are considerably faster to solve but hardly ever reach the optimal solution even if their relative deviation remains below 0.1%.

For larger instances, where the optimal solutions cannot be computed anymore in reasonable time, we ran CPLEX and applied a state-of-the-art evolutionary algorithm (EA) based on Raidl and Gottlieb [7] both with a time limit of 500 seconds. It turned out that both approaches yield consistently better solutions than the same algorithms executed on the complete instances which is confirmed by one-sided Wilcoxon rank tests with extremely low error probabilities. As can be expected, the restriction to the core problems allows the algorithms to enumerate more nodes of the branch and bound tree resp. perform more iterations of the EA and thus yields better solutions.

Combining the advantages of exact methods (CPLEX in our case) and the EA, we also constructed a collaborative framework running both algorithms in (quasi-) parallel. Information is exchanged by sending each other any detected new current best solution which is either incorporated into the population of the EA or used as a new lower bound by CPLEX. The EA uses greedy-type procedures to perform repair operations to reach feasibility after recombination or mutation and as a local improvement. Since these procedures are based on sorting by efficiencies, which are computed with the dual solutions of the LP-relaxation, these values are updated by the respective LP-solution of a branch and bound node whenever a new current best solution is found by CPLEX.

Extensive computational experiments produce in average the best results if this exchange of dual variables is indeed performed and if the running time is split by a 2:1 ratio between CPLEX and the EA (skewed cooperation). However, running the EA alone yields the highest number of best solutions over all instances but inferior solutions on some of them. Solving core problems with this collaborative approach leads to better solutions for small core sizes, but more or less equal performance on larger cores.

Finally, we compared the best algorithm we developed, namely the collaborative approach with skewed cooperation, exchange of dual variables and applied to a larger core problem, with the currently best heuristic by Vasquez and Vimont [8]. Their approach is based on a tabu search framework which partitions the solution space by a cardinality constraint that is iterated over all feasible numbers of items.

It turns out that for the largest benchmark instances with $n = 500$ our approach yields better solutions for $m = 5$ and solutions of roughly comparable quality for $m = 10$. For $m = 30$ we could not match the results in [8]. However, our approach was limited to 4 hours of running time whereas [8] consumes 16 hours in average (on a comparable machine) and up to 80 hours for the $m = 30$ instances.

A full paper on these investigations is available in [6] while some results were already published in [5]. Many more details can be found in Puchinger [4].

References

[1] P.C. Chu and J.E. Beasley, *A genetic algorithm for the multidimensional knapsack problem*, Journal of Heuristics **4** (1998), 63–86. benchmark instances at: `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/mknapinfo.html`

[2] A. Fréville, *The multidimensional 0-1 knapsack problem: An overview*, European Journal of Operational Research **155** (2004), 1–21.

[3] H. Kellerer, U. Pferschy and D. Pisinger, *Knapsack Problems*, Springer, 2004.

[4] J. Puchinger, *Combining Metaheuristics and Integer Programming for Solving Cutting and Packing Problems*, PhD thesis, Vienna University of Technology, 2006.

[5] J. Puchinger, G.R. Raidl and U. Pferschy, *The core concept for the multidimensional knapsack problem*, Proceedings of the 6th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP 06), LNCS **3906** (2006), 195–208.

[6] J. Puchinger, G.R. Raidl and U. Pferschy, *The multidimensional knapsack problem: Structure and Algorithms*, Technical Report TR-186-1-07-01, Vienna University of Technology, Department of Computer Science, 2007, submitted.

[7] G.R. Raidl and J. Gottlieb, *Empirical analysis of locality, heritability and heuristic bias in evolutionary algorithms: A case study for the multidimensional knapsack problem*, Evolutionary Computation **13** (2005), 441–475.

[8] M. Vasquez and Y. Vimont, *Improved results on the 0–1 multidimensional knapsack problem*, European Journal of Operational Research **165** (2005), 70–81.

Evolutionary Algorithms for Matroid Optimization Problems

Joachim Reichel
(joint work with Martin Skutella)

Evolutionary algorithms are widely used in practice, however their empirical good behavior is not well understood from a methodological point of view. Our goal is to understand what underlying problem structures are promising for evolutionary algorithms.

We consider two similar evolutionary algorithms, namely (1+1) EA and RLS. Both algorithms are initialized by a random element s from the search space $\{0,1\}^m$, generate another element s' and replace s by s' if the fitness of s' is better than that of s. The last two steps are repeated until some termination criterion is met. (1+1) EA obtains s' by flipping the bits of s u.a.r. with probability $1/m$ whereas RLS chooses one or two bits of s u.a.r. and flips these bits.

In the *minimum weight basis problem* the task is to find a basis of a given matroid $(E, \mathcal{F})$ that has minimum weight w.r.t. a weight function $w : E \mapsto \mathbb{N}$. This problem is a generalization of the minimum spanning tree problem. We proof that the expected number of generations until the considered evolutionary algorithms find a minimum weight basis is bounded by $O(|E|^2(\log r(E) + \log w_{\max}))$, where $r(E)$ denotes the rank of the matroid and $w_{\max}$ the maximum weight. This result is based on earlier work by Neumann and Wegener [1] who also proved a lower bound of $\Omega(|E|^2 \log r(E))$.

Another classic matroid problem is the *matroid intersection problem.* Here two matroids $(E, \mathcal{F}_1)$ and $(E, \mathcal{F}_2)$ are given and the task is to find a common independent set $X \in \mathcal{F}_1 \cap \mathcal{F}_2$ of maximum cardinality. This problem is a generalization of the bipartite matching problem. Evolutionary algorithms are able to compute a $(1-\epsilon)$-approximation of this problem within $O(|E|^{2\lceil 1/\epsilon \rceil})$ generations in expectation. This is an extension of earlier work by Giel and Wegener [2] who also proved an exponential lower bound to obtain the exact solution.

In the *weighted matroid intersection problem* an additional weight function $w : E \mapsto \mathbb{N}$ is given and the weight of the common independent set has to be maximized. Here the expected number of generations for a $\frac{1}{2}$-approximation is given by $O(|E|^4(\log r(E) + \log w_{\max}))$. This result can be extended to the intersection of $p \geq 3$ matroids, which is an NP-hard problem. In this case, the expected number of generations needed to obtain a $\frac{1}{p}$-approximation is bounded by $O(|E|^{p+2}(\log r(E) + \log w_{\max}))$. Note that the approximation ratio of $\frac{1}{p}$ is exactly the same ratio as that of the Greedy algorithm.

Our results show that problems whose underlying structure boils down to matroids can successfully be treated by evolutionary algorithms. See [3] for details.

References

[1] F. Neumann and I. Wegener, *Randomized local search, evolutionary algorithms and the minimum spanning tree problem*, Proc. of the 6th Genetic and Evolutionary Computation Conference (GECCO '04), Seattle, USA, 2004, 713–724.

[2] O. Giel and I. Wegener, *Evolutionary algorithms and the maximum matching problem*, Proc. of the 20th Symp. on Theoretical Aspects of Computer Science (STACS '03), 2003, 415–426.

[3] J. Reichel and M. Skutella, *Evolutionary Algorithms and Matroid Optimization Problems*, Proc. of the 9th Genetic and Evolutionary Computation Conference (GECCO '07), London, 2007, to appear.

Worst Case and Probabilistic Analysis of the 2-Opt Algorithm for the TSP

Heiko Röglin
(joint work with Matthias Englert and Berthold Vöcking)

2-Opt is probably the most basic and widely used local search heuristic for the TSP. This heuristic achieves amazingly good results on "real world" Euclidean instances both with respect to running time and approximation ratio. There are numerous experimental studies on the performance of 2-Opt. However, the theoretical knowledge about this heuristic is still very limited. Not even its worst case running time on Euclidean instances was known so far. We clarify this issue by presenting a family of Euclidean instances on which 2-Opt can take an exponential number of steps.

Previous probabilistic analyses were restricted to instances in which n points are placed uniformly at random in the unit square $[0,1]^2$, where it was shown that the expected number of steps is bounded by $\tilde{O}(n^{10})$ for Euclidean instances. We consider a more advanced model of probabilistic instances in which the points can be placed according to general distributions on $[0,1]^2$. In particular, we allow different distributions for different points. We study the expected running time in terms of the number n of points and the maximal density ϕ of the probability distributions. We show an upper bound on the expected length of any 2-Opt improvement path of $\tilde{O}(n^{4+1/3} \cdot \phi^{8/3})$. When starting with an initial tour computed by an insertion heuristic, the upper bound on the expected number of steps improves even to $\tilde{O}(n^{3+5/6} \cdot \phi^{8/3})$. In addition, we prove an upper bound of $O(\sqrt{\phi})$ on the expected approximation factor. Our probabilistic analysis covers as special cases the uniform input model with $\phi = 1$ and a smoothed analysis with Gaussian perturbations of standard deviation σ with $\phi \sim 1/\sigma^2$.

References

[1] B. Chandra, H.J. Karloff, and C.A. Tovey, *New results on the old k-Opt algorithm for the traveling salesman problem*, SIAM Journal on Computing, **28**(6) (1999), 1998–2029.

[2] M. Englert, H. Röglin, and B. Vöcking, *Worst case and probabilistic analysis of the 2-Opt algorithm for the TSP*, In Proc. of the 18th ACM-SIAM Symp. on Discrete Algorithms (SODA), 2007, 1295–1304.

[3] D. S. Johnson and L.A. McGeoch, *The traveling salesman problem: A case study in local optimization*, In E.H.L. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, John Wiley and Sons, 1997.

[4] D.A. Spielman and S.H. Teng, *Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time*, Journal of the ACM, **51**(3) (2004), 385–463.

Efficient Data Structure Libraries

Leonor Frias
(joint work with Jordi Petit and Salvador Roura)

1. Introduction

Modern software design relies on the interaction and extension of well-defined, robust, highly tuned and flexible predefined software components. This approach allows system designers to concentrate on the high level logic and so, save money and time.

Data structures libraries are software components that define interfaces and implement fundamental data structures and algorithms. Nowadays, they are included as part of most programming languages. We focus on the *Standard Template Library (STL)*, which is the algorithmic core of the C++ standard library [4]. From a theoretical point of view, the knowledge required to implement the STL is well laid down on basic textbooks on algorithms and data structures. In fact, current widely used STL implementations offered by compiler and library vendors are based on these.

In the last years, there have been great advances in computing technology, and together the computing needs have evolved with them. On the algorithm community, new and more realistic models of computation have been introduced to exploit the available technology. However, most of this research has not yet been ported to general use software libraries. Though it is true that the requirements in the *theoretical world* and in the *actual* libraries are often quite different, they are both dealing with the same abstractions.

2. Goal

Our goal is to enhance different STL components with the most up-to-date knowledge on algorithms and data structures while keeping with STL requirements.

Experimentation is the key tool. Nonetheless, a strong knowledge on the library, data structures in general and some hints on analysis of analysis of algorithms are required.

We aim to enhance data structures in the following senses. On the one hand, we want to extend the STL components capabilities while retaining the cost and functionality bounds of the rest of operations. On the other hand, we want to improve components performance using recent developments in the field. My most recent work has focused on performance.

Just following I present some specific topics in which I have been working.

3. Enhancing STL `map` with rank operations

A `map` in the STL corresponds to a dictionary abstract data type with the following constraints:

- elements must be sorted by key
- query/update operations by key must be logarithmic time
- sequential access with iterators must be provided (each step in amortized constant time)

However, operations that take into account the cardinal position of the element in the sorted sequence, such as *rank* or *ith*, are not considered. Therefore, *any* balanced binary search tree can be used to implement STL `map`s. What is more, most existing implementations are based on red-black trees.

If rank operations are to be considered, one could think on simply augmenting the tree. But instead, our approach uses *Logarithmic Binary Search Tree*(LBST) [6] because efficient rank/ith operations can be offered without extra fields. Furthermore, we have used standard iterators notation (*random access iterators*) to implement the new operations.

Our experiments have showed that our final implementation achieved the same or better performance as GCC implementation (based on red-black trees) for almost all operations, but with extra *functionality.* Note, experimentation was key to successively refine the implementation. Further, we have contributed a new variant of LBSTs that uses top-down insertion/erase operations.

More information on this work can be found in [1].

4. Improving cache performance of STL `lists`

Memory hierarchies try to minimize the gap between memory access time and arithmetic operation time. Their success relies on the common locality property of data and programs. However, many data structures (specially dynamic memory based) exhibit poor locality.

As a response, the algorithmic community proposes alternative data structures/algorithms that organize data such that the logical access pattern is similar to the physical memory locations. If cache parameters are used in the actual algorithm or data structure, the so-called *cache-aware* approach is followed. In contrast, if no cache parameters are used at all, a *cache-oblivious* [3] design is achieved.

Our work aims to fill the gap between *double-linked lists* implementations, that easily cope with standard requirements but have a poor cache locality, and their cache-conscious counterparts that are designed for very different requirements. Specifically, the main problem has been keeping both with STL `list`s iterator functionality and constant cost operations requirements. In particular, some of the main issues are that there can be an arbitrary number of iterators and that operations cannot invalidate them.

Our approach consists on a cache-aware double-linked list of *buckets.* This ensures locality inside the buckets, while logically consecutive buckets are let to be physically far. Moreover, all the iterators referred to an element are identified with a dynamic node (*relayer*) that points to it. Given that we expect an small number of iterators, the best solution is keeping the relayers on a sorted linked list. Nonetheless, the asymptotic costs hold whatever the number of iterators.

The experiments have showed that our approach is preferable in many (common) situations to classical double-linked list implementations. Further, our implementation are still competitive with (unusual) big load of iterators and bucket capacity is not a critical parameter.

More information on this work can be found in [2].

5. Parallel implementation of STL containers bulk operations

Multicore microprocessors consist on 2 or more independent processors into a single package (integrated circuit). Typically they have 1-2 levels of private cache and 1 level of shared memory.

Parallel algorithms and data structures is not a new field. However, with multicore computers parallelism is available at the fingertips of any user. This makes almost a *must* to take advantage of it. In particular, data structure libraries should take advantage of the new technology both offering a parallel implementation of the operations and supporting the parallel usage of the library components.

Currently, we are working jointly with J. Singler and P. Sanders from *Universität Karlsruhe* to implement STL containers bulk operations. See in [5] an overview of their work on the parallel implementation of the STL.

References

[1] L. Frias, *Extending STL maps using LBSTs*, Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALCO), 2005, SIAM, 155–166.

[2] L. Frias, J. Petit, and S. Roura, *Lists Revisited: Cache Conscious STL Lists*, Proceedings of the Fifth International Workshop on Experimental Algorithms (WEA), LNCS **4007** (2006), 121–133.

[3] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, *Cache-oblivious algorithms*, FOCS '99, 1999, IEEE Computer Society, 285.

[4] International Standard ISO/IEC 14882, *Programming languages — C++*, American National Standard Institute, 1st edition, 1998.

[5] Felix Putze, Peter Sanders, and Johannes Singler, *Mcstl: the multi-core standard template library*, PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, 2007, ACM.

[6] S. Roura, *A new method for balancing binary search trees*, in F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *28th International Colloquium on Automata, Languages and Programming*, LNCS **2076** (2001), 469–480.

Stronger Guarantees for Standard-Library Containers[1]

Jyrki Katajainen

The Standard Template Library (STL) [13, 14] is a library of generic algorithms and data structures that has been incorporated in the C++ standard [1] and ships with all modern C++ compilers. In the CPH STL project [4] our goal is to implement an enhanced edition of the STL. Initially, our focus was on time and space efficiency of the STL components, but now we are also focusing on safety, reliability, and usability of the components. In my talk, I briefly discussed four types of guarantees our library should be able to provide for its users: time optimality, iterator validity, exception safety, and space efficiency.

In the CPH STL, every container is a bridge class that calls the functions available in the actual realization given as a template argument. For example, the interface of a meldable priority queue looks as follows [11]:

```
template <                           // Types:
  typename E,                        // elements stored
  typename C = std::less<E>,         // comparator used in element comparisons
  typename A = std::alloctor<E>,     // allocator used for memory management
  typename R = cphstl::binary_heap<E, C, A> // underlying realization
>
class meldable_priority_queue;
```

Therefore, if one wants to specify precisely which realization is to be used in a particular instantiation, four template arguments must be given. If one is satisfied with the default settings, just one template argument specifying the type of elements stored has to be given. Different realizations available at the library can then provide different guarantees suitable for specific purposes.

"Time" optimality. The fundamental difference between the development of a generic library and the development of a normal algorithm library is that we operate with semi-algorithms, not with algorithms. Let F be a function or a primitive whose cost and realization are not known. A component C may depend on F even if F is unknown at development time of C. In an ideal situation, C should work well for all potential realizations of F. If this is the case, a semi-algorithm is said to be *primitive oblivious* with respect to F. Of course, *optimally primitive-oblivious* semi-algorithms are of particular interest. The concept of primitive obliviousness was implicitly defined in [9].

For a semi-algorithm the unspecified primitives can, for example, be element reads and writes, element comparisons, element constructions and destructions, and other primitives whose cost may vary unpredictably (like a branch). Also, a function argument and template argument can define such a primitive. If the

[1]Presented at the Algorithm Engineering meeting held in Oberwolfach on May 2007. Partially supported by the Danish Natural Science Research Council under contract 272-05-0272 (project "Generic programming—algorithms and tools").

CPH STL Report 2007-3, May 2007.

primitives are reads and writes, we get that the concept of cache obliviousness is a special case of the concept of primitive obliviousness. If the primitive is element comparison, we get a link back to the classical comparison complexity. However, since the cost of individual element comparisons may vary (cf. integer comparisons vs. string comparisons) it can be difficult to develop semi-algorithms that are optimally primitive oblivious with respect to element comparisons.

The standard technique used in generic libraries is to provide several specializations of a component for some specific data types. However, since there is an infinite number of data types that could be given as a template argument, it is impossible to provide all potential specializations in a program library. It would be nice if a component could be made primitive oblivious, or even optimally primitive oblivious, with respect to all unspecified primitives at the same time. Next we will describe one such optimally primitive-oblivious semi-algorithm to show that for some problems such semi-algorithms exist.

In the *0/1-sorting problem*, we are given a sequence S of elements drawn from a universe $\mathcal{E}$ and a characteristic function $f\colon \mathcal{E} \to \{0,1\}$. We call an element x zero, if $f(x) = 0$, and one, if $f(x) = 1$. Now the task is to rearrange the elements in S so that every zero is placed before any one. Moreover, this reordering should be done *stably* without altering the relative order of elements having the same f-value. In the STL, function `stable_partition()` is designed to used for 0/1-sorting.

A trivial semi-algorithm can solve this problem as follows: Scan the input sequence S twice, move first zeros and then ones to a temporary area, and copy the elements back to S. Each element is read and written $O(1)$ times and only sequential access is involved. That is, this semi-algorithm is optimally cache oblivious. Additionally, each element is copied $O(1)$ times and, for each element, characteristic function f is evaluated $O(1)$ times. That is, there exists a semi-algorithm for 0/1-sorting that is optimally primitive oblivious with respect to all unspecified primitives. The problem turns out to be much harder, as discussed in [9], if 0/1-sorting should be done in-place using only $O(1)$ words of extra space. The development of other optimally primitive-oblivious semi-algorithms is left for an interested reader. Natural candidates would be the other components specified in the STL.

Iterator validity. A *locator* (a term adopted from [10]) is a mechanism for maintaining the association between an element and its location in a data structure. Technically, locators are objects that can be created, destroyed, copied, compared, and they provide access to the element stored at the location pointed to. An *iterator* is a generalization of a locator that captures the concepts location and iteration in a container of elements. For example, a *bidirectional iterator* is a locator that also supports the operations `++` and `--` allowing one to access the elements next to the current location.

A data structure provides *iterator validity* if the iterators to the compartments storing the elements are kept valid at all times. Of the standard containers only lists and associative containers are required to keep their iterators valid. For other containers, in the C++ standard there are precise rules stating which iterators are kept valid by which operations in which circumstances. For a programmer, it can be difficult to remember these kinds of rules.

For many data structures, iterator validity can be achieved by storing handles to the elements instead of the elements themselves. The technique of using handles is described in the textbook by Cormen et al. [3, Section 6.5]. In the CPH STL project this approach has been used to realize iterator-valid dynamic arrays. Normally, the extra indirection has a small performance penalty [8], but since one can loose the spatial locality of elements, in worst-case scenarios the cache behaviour gets worse. Our target is to provide an iterator-valid realization for each container class. In most cases an iterator-valid realization requires a bit more memory compared to a straightforward realization.

Exception safety. An operation on an object is said to be *exception safe* if that operation leaves the object in a valid state when the operation is terminated by throwing an exception [16, Appendix E]. A *valid state* means a state that allows the object to be accessed and destroyed without causing undefined behaviour or an exception to be thrown from a destructor. In addition, the operation should ensure that every resource that it acquired is (eventually) released.

In the C++ standard, different guarantees provided by a container operation are classified in four categories [16, Appendix E]:

No guarantee: If an exception is thrown, any container being manipulated is possibly corrupted.

Strong guarantee: If an exception is thrown, any container being manipulated remains in the state in which it was before the operation started. Think of *roll-back semantics* for database transactions!

Basic guarantee: The basic invariants of the containers being manipulated are maintained, and no resources are leaked.

Nothrow guarantee: In addition to the basic guarantee, the operation is guaranteed not to throw an exception.

Normally, container operations are known to provide the basic guarantee, but in some special cases stronger exception-safety guarantees can be given. A programmer has to consult the documentation to recall the exact rules.

In general, all user-supplied functions and template arguments can throw an exception. As an example let us consider the copy constructor for a set:

```
template ⟨typename E, typename C, typename A⟩
set⟨E, C, A⟩::set(const set&);
```

In this particular case, the following operations can throw an exception:

- function `allocate()` of the allocator (of type `A`) indicating that no memory is available,
- copy constructor of the allocator,
- copy constructor of the element (of type `E`) used by function `construct()` of the allocator,
- invocation of the comparator (of type `C`), and
- copy constructor of the comparator.

On the other hand, any primitive operation for the following types cannot throw an exception:

- built-in types,
- types without user-defined operations,
- classes with operations that do not throw, and
- functions from the C library unless they take a function argument that does.

Basically, all classes with destructors that do not throw and which can be easily verified to leave their operands in valid states are friendly for library writers. It is the responsibility of library users to ensure that destructors do not throw exceptions. Without this assumption it would be difficult to write library code that would provide the strong guarantee of exception safety.

It has turned out to be difficult to write exception-safe code (cf. [16, p. 943]) so our current prototypes do not yet provide strong exception safety. In theory, there is no asymptotic efficiency penalty, just more (a lot more) careful programming is required. Also testing, whether your code is exception safe or not, is tedious. So far we have done this by visual code inspection. It should be pointed out that exception-safe components cannot be easily combined. That is, there are some fundamental problems, which are not algorithmic, that have to be solved to make exception-safe programming easier.

Space efficiency. In the C++ standard [1] no explicit space bounds are specified. In widely distributed implementations, like the SGI STL [15], the amount of space used by all element containers is linear, except that for the dynamic-array class the allocated memory is freed only at the time of destruction. In the CPH STL if a data structure stores n elements, it is required to use at most $O(n)$ extra space (or less if possible). Examples of highly space-efficient data structures that have been devised in the CPH STL project include dynamic arrays [12], dictionaries and priority queues [2]; the memory overhead is $O(\sqrt{n})$ words and elements for dynamic arrays, and $(1 + \varepsilon)n$ words for dictionaries and priority queues.

Conclusions. In the CPH STL project our focus is not only on time and space efficiency, but also on safety, reliability, and usability. Ideally, the library should offer off-the-shelf components that can be used in a plug-and-play manner and that provide raw speed, iterator validity, exception safety, and space efficiency. Based on the work with my students—and the complicated programming errors experienced by them—I firmly believe that safe and reliable components are warmly welcomed by many programmers.

We have been able to devise several data structures whose performance is close to absolute minimum with respect to the number of element comparisons performed [5, 6, 7], but unfortunately these data structures are complex and not as such suited for a practical implementation. To reveal the practical relevance of the ideas presented, algorithm engineering will be necessary.

The development of generic libraries is challenging (as can be their use, especially, because of long and incomprehensible error messages provided by contemporary compilers). There are several theoretical and practical challenges to be taken. In my talk some of the areas requiring further work were identified. Some of the open questions are not algorithmic so these should be solved together with experts working in other areas of computing.

Finally, I would like to make you aware that, if you have developed an industry-strength STL component and feel that it should be a part of a program library, you are welcome to donate your code to the CPH STL. We promise to consider all donations seriously. When released, the library will be placed in the public domain. Even after a release, a donor will share the copyright of his or her code with the Performance Engineering Laboratory at the University of Copenhagen.

Acknowledgements. I enjoyed the stay at *Mathematisches Forschungsinstitut Oberwolfach* and I thank the organizers for inviting me to this meeting on algorithm engineering.

References

[1] British Standards Institute, *The C++ Standard: Incorporating Technical Corrigendum 1*, BS ISO/IEC 14882:2003 (2nd Edition), John Wiley and Sons, Ltd., 2003.

[2] H. Brönnimann, J. Katajainen, and P. Morin, *Putting your data structure on a diet*, CPH STL Report **2007-1**, Department of Computing, University of Copenhagen, 2007.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, The MIT Press, 2001.

[4] Department of Computing, University of Copenhagen, *The CPH STL*, Website accessible at `http://www.cphstl.dk/`, 2000–2007.

[5] A. Elmasry, C. Jensen, and J. Katajainen, *A framework for speeding up priority-queue operations*, CPH STL Report **2004-3**, Department of Computing, University of Copenhagen, 2004.

[6] A. Elmasry, C. Jensen, and J. Katajainen, *Two-tier relaxed heaps*, Proceedings of the 17th International Symposium on Algorithms and Computation, LNCS **4288** (2006), 308–317.

[7] A. Elmasry, C. Jensen, and J. Katajainen, *Two new methods for transforming priority queues into double-ended priority queues*, CPH STL Report **2006-9**, Department of Computing, University of Copenhagen, 2006.

[8] N. Esbensen, *The cost of iterator validity*, Proceedings of the 6th STL Workshop, CPH STL Report **2006-9**, Department of Computing, University of Copenhagen, 2006, 34–44.

[9] G. Franceschini and J. Katajainen, *Generic algorithm for 0/1-sorting*, CPH STL Report **2006-5**, Department of Computing, University of Copenhagen, 2006.

[10] M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, John Wiley & Sons, Inc., 1998.

[11] J. Katajainen, *Project proposal: A meldable, iterator-valid priority queue*, CPH STL Report **2005-1**, Department of Computing, University of Copenhagen, 2005.

[12] J. Katajainen and B. B. Mortensen, *Experiences with the design and implementation of space-efficient deques*, Proceedings of the 5th Workshop on Algorithm Engineering, LNCS **2141** (2001), 39–50.

[13] D. R. Musser and A. A. Stepanov, *Algorithm-oriented generic libraries*, Software—Practice and Experience **24**(7) (1994), 623–642.

[14] P. J. Plauger, A. A. Stepanov, M. Lee, and D. R. Musser, *The C++ Standard Template Library*, Prentice Hall PTR, 2001.

[15] Silicon Graphics, Inc., *Standard template library programmer's guide*, Website accessible at `http://www.sgi.com/tech/stl/`, 1993–2006.

[16] B. Stroustrup, *The C++ Programming Language*, Special edition, Addison-Wesley, 2000.

Algorithm Engineering: Some Challenges in Computer Graphics

Friedhelm Meyer auf der Heide
(joint work with Matthias Fischer)

We identify some specific problems occurring in Computer Graphics, especially for rendering and walkthrough, which makes experimental and theoretical runtime analysis very complicated:

- The underlying machine model has to be extended by the properties of the graphics hardware, especially their capability to execute very fast, hardware-supported hidden surface removal.
- Further, input complexity is not sufficiently characterised by the number of polygons the scene consists of. Rather, we have to take into account viewpoint-dependent parameters like projected area of all, also occluded, polygons.

As the main contribution we present strategies to reduce the complexity of the part of the scene that is sent to the graphics hardware. First we present two approximation algorithms. They might allow a few pixel errors, but reduce the influence of the number of triangles on the runtime from linear to logarithmic, based on two random sampling approaches. The time bounds are supported both theoretically and experimentally. Then we present the problem of deciding whether occlusion culling is worthwhile by introducing the aspect graph and a heuristic strategy.

This is joint work with Matthias Fischer, based, among others, on the Randomized z-Buffer Algorithm (Fischer, MadH, Peter, Straßer, Wand, SIGGRAPH 01) and the SampleTree (Fischer, Krokowski, Klein, MadH, Wand, Wanka, PRESENCE 04).

The Importance of Experiments in Game Theory via some Case Studies

Paul G. Spirakis
(joint work with Panagiota N. Panagopoulou)

The most important solution concept in game theory is the notion of Nash equilibrium [3]. Despite its certain existence, the problem of computing a Nash equilibrium was proved to be complete in PPAD [1]. In view of this fact, the experimental study of games and the simulation of the selfish behavior of the players involved can help in understanding how an equilibrium is reached.

We focus on the experimental study of weighted, single commodity network congestion games [4]. In such games, each one of n selfish agents wishes to route her load over a single-source and single-destination network G so as to minimize the total delay experienced on the path she chooses. We assume that individual link delays are equal to the total load of the link.

In [2] it was shown that such games possess pure Nash equilibria and suggested an algorithm (which is in fact a potential-based method) for computing a pure Nash equilibrium. This algorithm converts any given non-equilibrium allocation into a pure Nash equilibrium by performing a sequence of *greedy selfish steps*: a greedy selfish step is an agent's change of her current pure strategy (i.e. path) to her best pure strategy with respect to the current allocation of all other users. A superficial analysis of this algorithm gives an upper bound on its time which is polynomial in n and the sum of the agents' loads W, and this bound can be exponential in n when some weights are exponential.

However, in [4] we provide strong experimental evidence that this algorithm actually converges to a pure Nash equilibrium in *polynomial time*. More specifically, our experimental findings suggest that the running time is a polynomial function of n and $\log W$ for a variety of network topologies (e.g. grid, clique, trees etc) and distributions of the agents' loads.

Moreover, our experimental evaluation shows that the initial allocation of agents to paths significantly affects the convergence time of the algorithm. In particular, we suggest an initial allocation (the *shortest path allocation*) that dramatically accelerates this algorithm and is in fact a few greedy selfish steps far from a pure Nash equilibrium. While an arbitrary initial allocation does not assure a similarly fast convergence, the algorithm terminates in polynomial time for this case as well.

In addition, our experimental study suggests that the worst-case input for an arbitrary initial allocation occurs when all agents' loads are distinct and some of them are exponential.

References

[1] X. Chen and X. Deng, *Settling the complexity of 2-player Nash equilibrium*, Proceedings of the 47th IEEE Symposium on Foundations of Computer Science (FOCS 2006), 261–272.

[2] D. Fotakis, S. Kontogiannis and P. Spirakis, *Selfish unsplittable flows*, Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004), 593–605.

[3] J. Nash, *Noncooperative games*, Annals of Mathematics, **54** (1951), 289–295.

[4] P. Panagopoulou and P. Spirakis, *Algorithms for pure Nash equilibria in weighted congestion games*, ACM Journal of Experimental Algorithmics, **11** (2006), Article No 2.7.

Optimal Resilient Dynamic Dictionaries

Rolf Fagerberg

(joint work with Gerth Stølting Brodal, Allan Grønlund Jørgensen, Gabriel Moruz, and Thomas Mølhave)

A wide palette of factors, such as power failures, radiation, and cosmic rays, have a harmful effect on the reliability of contemporary memory devices, causing *soft memory errors* [18, 19]. In a soft memory error, a bit flips and consequently the content of the corresponding memory cell gets corrupted. As storage technology develops, memory devices get smaller and more complex, and work at lower voltages and higher frequencies [6]. All these improvements increase the likelihood of soft memory errors, hence the rate of soft memory errors is expected to increase for both SRAM and DRAM memories [18]. Memory corruptions are of particular concern for applications dealing with massive amounts of data, e.g. search engines, since the large number of memory devices used to manipulate the data vastly increases the frequency of memory corruptions. Taking into account that the amount of cosmic rays increases with altitude, soft memory errors are of special interest in fields like avionics and space research.

Since most software assume a reliable memory, soft memory errors can be exploited to produce severe malfunctions, such as breaking cryptographic protocols [4, 20], taking control of a Java Virtual Machine [10], or breaking smart-cards and other security processors [1, 2, 17]. In particular, corrupted memory cells can have serious consequences for the output of algorithms. For instance, during binary search in a sorted array, a single corruption in the early stages of the algorithm can cause the search path to end as far as $\Omega(n)$ locations from its correct position.

Memory corruptions have been addressed in various ways, both at the hardware and software level. At the hardware level, they are tackled using error detection mechanisms, such as redundancy, parity checking, or Hamming codes. However, adopting such mechanisms involves non-negligible penalties with respect to performance, size, and cost, and therefore memories implementing them are rarely found in large scale clusters or ordinary workstations. At the software level, several different low-level techniques are used, such as algorithm based fault tolerance [11], assertions [16], control flow checking [21], or procedure duplication [14]. However, most of these handle instruction corruptions rather than data corruptions.

Dealing with unreliable information has been addressed in the algorithmic community in a number of settings. The liar model focuses on algorithms in the comparison model where the outcome of a comparison is possibly a lie. Several fundamental algorithms in this model, such as sorting and searching, have been proposed [5, 12, 15]. In particular, searching in a sorted sequence takes $O(\log n)$ time, even when the number of lies is proportional to the number of comparisons [5]. A standard technique used in the design of algorithms in the liar model is query replication. Unfortunately, this technique is not of much help when memory cells, and not comparisons, are unreliable.

Aumann and Bender [3] proposed fault-tolerant (pointer-based) data structures. To incur minimum overhead, their approach allows a certain amount of data, expressed as a function of the number of corruptions, to be lost upon pointer corruptions. In their framework memory faults are detectable upon access, i.e. trying to access a faulty pointer results in an error message. This model is not always appropriate, since in many practical applications the loss of valid data is not permitted. Furthermore, a pointer can get corrupted to a valid address and therefore an error message is not issued upon accessing it.

Finocchi and Italiano [9] introduced the *faulty-memory RAM.* In this model, memory corruptions occur at any time and at any place during the execution of an algorithm, and corrupted memory cells cannot be distinguished from uncorrupted cells. Motivated by the fact that registers in the processor are considered uncorruptible, $O(1)$ safe memory locations are provided. The model is parametrized by an upper bound, δ, on the number of corruptions occurring during the lifetime of an algorithm. An algorithm is resilient if it works correctly, at least on the set of uncorrupted cells in the input. In particular, a resilient searching algorithm returns a positive answer if there exists an uncorrupted element in the input equal to the search key. If there is no element, corrupted or uncorrupted, matching the search key, the algorithm returns a negative answer. Otherwise, the answer can be both positive and negative.

Several problems have been addressed in the faulty-memory RAM. In the original paper [9], lower bounds and (non-optimal) algorithms for sorting and searching were given. In particular, it was proved that searching in a sorted array takes $\Omega(\log n + \delta)$ time, i.e. it tolerates up to $O(\log n)$ corruptions while still preserving the classical $O(\log n)$ searching bound. Matching upper bounds for sorting and randomized searching, as well as a $O(\log n + \delta^{1+\epsilon})$ deterministic searching algorithm, were then given in [7]. Recently, resilient search trees that support searches, insertions, and deletions in $O(\log n + \delta^2)$ amortized time [8] were introduced. Finally, in [13] it was empirically shown that resilient sorting algorithms are of practical interest.

Our results. We propose two optimal resilient static dictionaries, a randomized one and a deterministic one, as well as a dynamic dictionary.

Randomized static dictionary: We introduce a resilient randomized static dictionary that support searches in $O(\log n + \delta)$ time, matching the bounds for randomized searching in [7]. We note however that our dictionary is somewhat simpler and uses only $O(\log \delta)$ worst case random bits, whereas the algorithm in [7] uses expected $O(\log \delta \cdot \log n)$ random bits.

Deterministic static dictionary: We give the first optimal resilient deterministic dictionary. It supports searches in a sorted array in $O(\log n + \delta)$ time in the worst case, matching the lower bounds from [9].

Dynamic dictionary: We extend this result to a dynamic dictionary supporting searches in $O(\log n + \delta)$ worst case time, and insertions and deletions in $O(\log n + \delta)$ amortized time. Also, it supports range queries in $O(\log n + \delta + k)$ worst case time, where k is the output size.

References

[1] R. Anderson and M. Kuhn, *Tamper resistance - a cautionary note*, Proc. 2nd Usenix Workshop on Electronic Commerce, 1996, 1–11.

[2] R. Anderson and M. Kuhn, *Low cost attacks on tamper resistant devices*, International Workshop on Security Protocols, 1997, 125–136.

[3] Y. Aumann and M. A. Bender, *Fault tolerant data structures*, Proc. 37th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, 1996, 580–589.

[4] D. Boneh, R. A. DeMillo, and R. J. Lipton, *On the importance of checking cryptographic protocols for faults*, Eurocrypt, 1997, 37–51.

[5] R. S. Borgstrom and S. R. Kosaraju, *Comparison-based search in the presence of errors*, Proc. 25th Annual ACM symposium on Theory of Computing, 1993, 130–136.

[6] C. Constantinescu, *Trends and challenges in VLSI circuit reliability*, IEEE micro, **23**(4) (2003), 14–19.

[7] I. Finocchi, F. Grandoni, and G. F. Italiano, *Optimal resilient sorting and searching in the presence of memory faults*, Proc. 33rd International Colloquium on Automata, Languages and Programming, LNCS **4051** (2006), 286–298.

[8] I. Finocchi, F. Grandoni, and G. F. Italiano, *Resilient search trees*, Proc. 18th ACM-SIAM Symposium on Discrete Algorithms, 2007, 547–554.

[9] I. Finocchi and G. F. Italiano, *Sorting and searching in the presence of memory faults (without redundancy)*, Proc. 36th Annual ACM Symposium on Theory of Computing, 2004, ACM Press, 101–110.

[10] S. Govindavajhala and A. W. Appel, *Using memory errors to attack a virtual machine*, IEEE Symposium on Security and Privacy, 2003, 154–165.

[11] K. H. Huang and J. A. Abraham, *Algorithm-based fault tolerance for matrix operations*, IEEE Transactions on Computers, **33** (1984), 518–528.

[12] K. B. Lakshmanan, B. Ravikumar, and K. Ganesan, *Coping with erroneous information while sorting*, IEEE Transactions on Computers, **40**(9) (1991), 1081–1084.

[13] U. F. Petrillo, I. Finocchi, and G. F. Italiano, *The price of resiliency: a case study on sorting with memory faults*, Proc. 14th Annual European Symposium on Algorithms, 2006, 768–779.

[14] D. K. Pradhan, *Fault-tolerant computer system design*, Prentice-Hall, Inc., 1996.

[15] B. Ravikumar, *A fault-tolerant merge sorting algorithm*, Proc. 8th Annual International Conference on Computing and Combinatorics, 2002, 440–447.

[16] M. Z. Rela, H. Madeira, and J. G. Silva, *Experimental evaluation of the fail-silent behaviour in programs with consistency checks*, Proc. 26th Annual International Symposium on Fault-Tolerant Computing, 1996, 394–403.

[17] S. P. Skorobogatov and R. J. Anderson, *Optical fault induction attacks*, Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems, 2002, 2–12.

[18] Tezzaron Semiconductor, *Soft errors in electronic memory - a white paper*, `http://www.tezzaron.com/about/papers/papers.html`, 2004.

[19] A. J. van de Goor, *Testing Semiconductor Memories: Theory and Practice*, ComTex Publishing, Gouda, The Netherlands, 1998, ISBN 90-804276-1-6.

[20] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer, *An experimental study of security vulnerabilities caused by errors*, Proc. International Conference on Dependable Systems and Networks, 2001, 421–430

[21] S. S. Yau and F.-C. Chen, *An approach to concurrent control flow checking*, IEEE Transactions on Software Engineering, **6**(2) (1980), 126–137.

Routing in Graphs with Applications to Real Time Material Flow Problems

Rolf H. Möhring
(joint work with Ewgenij Gawrilow, Ekkehard Köhler, Ines Spenke, and Björn Stenzel)

In modern logistic systems Automated Guided Vehicles (AGVs) are used for transportation tasks. An appropriate control of these AGVs is crucial for efficient transportation. They need to be assigned collision free routes in such a way that the throughput of goods is maximized. The determination of these routes is an online routing problem (nothing is known about future requests) and also a real-time problem, because fast answers are required (should be less than one second in practice).

We present an algorithm for this problem which avoids collisions, deadlocks, livelocks and other conflicts already at the time of route computation (conflict-free routes). We thus extend approaches of Huang, Palekar and Kapoor [3] and Kim and Tanchoco [4], respectively. In particular, we take physical properties of the AGVs into consideration in a more exact and flexible way.

The time dependent behavior of the AGVs is modeled by time-windows on the arcs of the routing graph(implicit time-expansion). Each time-window represents a free time slot at the corresponding arc depending on the routes of the AGVs that are already computed (see Fig. 1). The real-time computation for each routing request consists of the determination of a shortest path with time-windows (routing) and a subsequent readjustment of the time-windows (blocking).

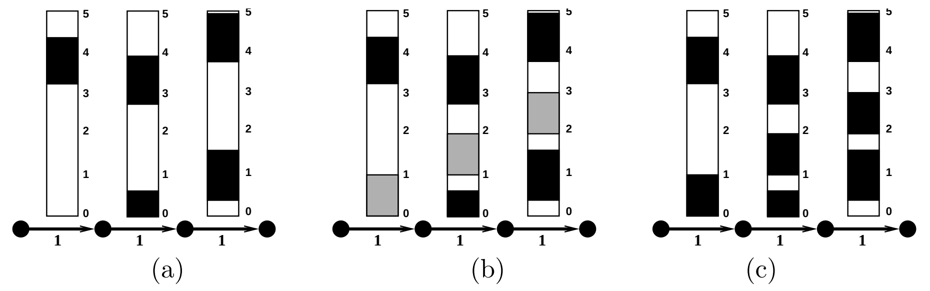

Figure 1. Real-time computation. (a) shows the situation before the new request arrives. There is a graph with some blockings (black) and some time-windows (white). The task is to compute a quickest path that respects the time-windows. This is illustrated in (b). The chosen path is blocked afterwards (see (c)).

The Shortest Path Problem With Time-Windows is NP-hard in general [2], but in this special case where cost correlates with elapsed time (travel time plus waiting time), our algorithm solves the problem in polynomial time (in the size of the graph and the number of time-windows) using a generalized Dijkstra algorithm algorithm that maintains an interval in each label (the expansion of such a label is shown in Fig. 2). In contrast, we can show in simplified model with constant travel times that the problem without waiting is weakly NP-hard, while the multicommodity case turns out to be strongly NP-hard [5]. This is related to the complexity of packet routing investigated in [1].

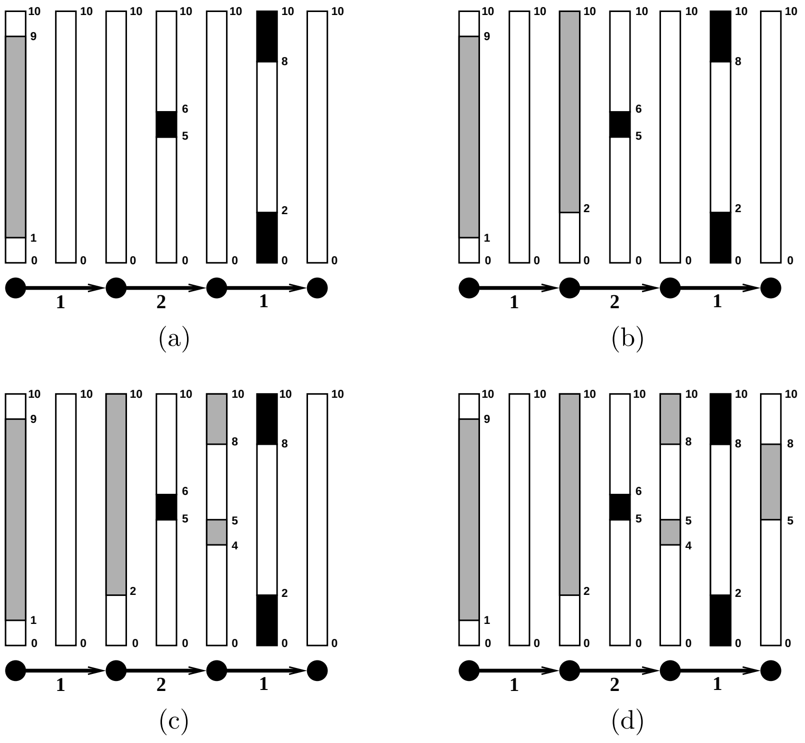

FIGURE 2. Label Expansion in the generalized Dijkstra algorithm. The label intervals are represented by gray bars (nodes). The blockings are colored black (arcs). The white intervals between these blockings are the time-windows. The figures (a) to (d) show the successive expansion of the label intervals.

Our routing algorithm shows very good computational times in practice and can also handle additional features such as a prescribed orientation of the AGVs at their destination. On a network with more than 30.000 arcs and up to 100 AGVs routing and blocking together take not more than some hundredth of a second on the average[1] (see Table 1).

In comparison with a static routing approach, in which collision avoidance is done at run time and not at route computation time, our algorithm shows a clear

[1]Hardware: AMD-Athlon 2100+ (1,7 Mhz) with 512 MB RAM.

Table 1. Computational times (in seconds).

Scenarios	Comp. per request		Search		Readjustment	
	maximal	∅	maximal	∅	maximal	∅
25-1G-L (25 AGVs)	0.35	0.10	0.32	0.08	0.04	0.02
25-1G-S (25 AGVs)	0.14	0.06	0.11	0.04	0.03	0.02
25-2G-L (25 AGVs)	0.24	0.06	0.24	0.05	0.03	0.01
25-2G-S (25 AGVs)	0.25	0.06	0.24	0.05	0.02	0.01
25-3G-L (25 AGVs)	0.29	0.06	0.27	0.05	0.04	0.01
25-3G-S (25 AGVs)	0.23	0.06	0.18	0.05	0.04	0.01
25-4G-L (25 AGVs)	0.18	0.04	0.16	0.03	0.03	0.01
25-4G-S (25 AGVs)	0.18	0.05	0.16	0.04	0.02	0.01
50-1G-L (50 AGVs)	0.35	0.10	0.31	0.08	0.04	0.02
50-1G-S (50 AGVs)	0.23	0.07	0.20	0.05	0.04	0.02
50-2G-L (50 AGVs)	0.32	0.06	0.30	0.05	0.04	0.01
50-2G-S (50 AGVs)	0.16	0.06	0.13	0.04	0.04	0.01
100G-L (100 AGVs)	0.26	0.06	0.23	0.05	0.05	0.01
100G-S (100 AGVs)	0.23	0.06	0.20	0.04	0.05	0.01

advantage w.r.t. total travel time for high traffic densities. In addition, it can also cope with unforeseen events occurring at run time and reroute AGVs in real time.

References

[1] C. Busch, M. Magdon-Ismail, M. Mavronicolas, P. Spirakis, *Direct routing: Algorithms and complexity*, in Proceedings of the 12th European Symposium on Algorithms (ESA'04), LNCS **3221** (2004), Springer, 134–45.

[2] Desrosiers et al., *Methods for routing with time windows*, European Journal of Operational Research **23** (1986), 236–245.

[3] J. Huang, U.S. Palekar, S. Kapoor, *A labeling algorithm for the navigation of automated guided vehicles*, Journal of engineering for industry **115** (1993), 315–321.

[4] Ch.W. Kim, J.M.A. Tanchoco, *Conflict-free shortest-time bidirectional AGV routing*, International Journal of Production Research **29**(12) (1991), 2377–2391.

[5] I. Spenke, *Complexity and approximation of static k-splittable flows and dynamic grid flows*, PhD Thesis, Technische Universität Berlin, 2006.

The Price of Resiliency: A Case Study on Sorting with Memory Faults

Irene Finocchi

(joint work with Umberto Ferraro-Petrillo, Fabrizio Grandoni, and Giuseppe F. Italiano)

The inexpensive memories used in today's computer platforms are not fully safe, and sometimes the content of a memory unit may be temporarily or permanently lost or damaged. This may depend on manufacturing defects, power failures, or environmental conditions such as cosmic radiation and alpha particles. Unfortunately, even very few memory faults may jeopardize the correctness of the underlying algorithms: for instance, if we want to search for a key in a sorted sequence subject to memory faults, corrupted keys may lead the search in the wrong direction.

The quest for reliable computation in unreliable memories arises in an increasing number of different settings, including large-scale applications that require the processing of massive data sets and fault-based cryptanalysis. In the design of reliable systems, when specific hardware for fault detection and correction is not available or it is too expensive, it makes sense to look for a solution to these problems at the application level, i.e., to design algorithms and data structures that are able to perform the tasks they were designed for, even in the presence of unreliable or corrupted information. Informally, we have a *memory fault* when the correct value that should be stored in a memory location gets altered because of a failure. We say that an algorithm or a data structure is *resilient* to memory faults if, despite the corruption of some memory values during its lifetime, it is nevertheless able to produce a correct output (at least) on the set of uncorrupted values.

In the talk we address the design of resilient sorting algorithms. Since we do not want to exploit data replication, our algorithms do not wish to recover corrupted data, but simply to be correct on uncorrupted data, without incurring any time or space overhead. More formally, we can state the resilient sorting problem as follows:

> *Resilient sorting*: we are given a set of n keys that need to be sorted. The values of some keys may be arbitrarily corrupted during the sorting process. The problem is to order correctly the set of uncorrupted keys.

This is the best that we can achieve in the presence of memory faults: we cannot indeed prevent keys corrupted at the very end of the algorithm execution from occupying wrong positions in the output sequence.

We first show how to sort resiliently in $O(n \log n + \delta^3)$ time, where n is the number of keys to be sorted and δ is an upper bound on the number of faults that can happen throughout the execution of the algorithm. We then experimentally investigate the impact of memory faults both on the correctness and on the running times of sorting algorithms. To achieve this goal, we develop a software testbed that simulates different fault injection strategies, and perform a thorough experimental study using a combination of several fault parameters. Our experiments give evidence that simple-minded approaches to the resilient sorting problem are largely impractical, while the design of more sophisticated algorithms seems really worth the effort.

Bandwidth Minimization: Human Stupidity still Beats Artificial Intelligence

Alberto Caprara
(joint work with Juan-José Salazar-González)

The *bandwidth* problem, calling for a linear layout of a graph in which the maximum distance between adjacent nodes is minimized, is a classical problem in combinatorial optimization that finds its main application in the solution of systems of linear equations. Specifically, finding a minimum bandwidth layout of a graph is the same as permuting the rows and columns of a symmetric square matrix so as to minimize the distance of the nonzero elements from the main diagonal, and systems of linear equations are easier to solve if this distance is small.

One of the main characteristics of the bandwidth problem is that the methods which are successful for other important combinatorial optimization problems do not seem to be particularly useful for its solution. For instance, preliminary computational experiments indicated that some natural integer linear programming formulations of the problem have extremely weak linear programming relaxations. A well-known strong lower bound on the minimum bandwidth of a graph introduced in the early 70s [2] is the so-called *density* lower bound, that was shown to be on average almost equal to the minimum bandwidth (under an appropriate distribution of the instances) [4]. On the other hand, it is unclear how to compute this bound as well as how to use it within an enumerative algorithm to fathom subproblems for which part of the solution is fixed (typically, the first/last nodes in the layout). It is also known that the problem can be solved in polynomial time by dynamic programming if the bandwidth is bounded by a constant [5], but the exponent of the polynomial for both the time and the space complexity is given by the bandwidth itself, which makes the method impractical when this value is not very small.

As a consequence of the above situation, although successful ad hoc heuristics have been proposed for the problem, no exact algorithms nor strong lower bounding procedures capable of tackling instances of reasonable size can be found in the literature. The only recent exception is [3], that illustrates an enumerative algorithm implicitly based on a very simple lower bound that can be computed in a very effective way. This algorithm works very well for dense graphs, essentially because for these graphs the simple bound coincides with other stronger bounds, whereas it is not effective for sparse instances as those that are encountered in practice. Note that the optimal bandwidth value is unknown for most instances in the "Matrix Market" collection [1], which is the main source of instances for the problem.

Although spending long time to minimize the bandwidth of a single system of linear equations is certainly not worth to speed-up the solution of the system itself, finding the exact value of the bandwidth for a relevant set of test instances is a challenging optimization problem in itself, and its outcome can be used to test the effectiveness of practical heuristics. Moreover, there are cases in which many systems of linear equations with the same nonzero pattern must be solved, in which case it is worth investing time in the corresponding (unique) bandwidth instance since this is going to speed-up the solution of all the systems.

In this work, we first analyze some lower bounds on the minimum bandwidth, discussing the complexity of their computation and some dominance relations among them. Specifically, we show that the computation of the density lower bound is NP-hard, and introduce a new lower bound that can be found efficiently and is suited for use within an enumerative scheme. Interpreting this latter lower bound as the optimal value of a suitable integer linear programming relaxation of the problem leads naturally to a stronger bound to be used within enumeration, still efficient to compute. We stress that in this relaxation variables are restricted to be *integer* (if this condition was relaxed, the model would be useless). We further tighten this relaxation, showing that a better lower bound can be found within short time by solving a suitable *bilevel* integer linear program. The resulting branch-and-bound method is quite successful, in that it is capable of solving to proven optimality 24 out of the 30 instances from the literature associated with graphs having up to 200 nodes, each within less than one minute, whereas only the 10 easiest instances can be solved by the method of [3]. We also show how our method behaves for random instances, confirming its effectiveness when the graph is sparse. Finally, we devise a simple but effective method to compute the density lower bound, whose value is reported for all instances in the literature with up to 250 nodes except one.

A closer look at the lower bounds used within enumeration that are mentioned above shows that their computation boils down to testing if it is feasible to assign different integer values to a set of variables subject to lower and upper bound constraints on each value. Although this is what one would expect state-of-the-art constraint programming solvers to be able to do automatically and effectively, the solvers that we tested failed miserably even on some toy bandwidth instances, that are on the other hand solvable within negligible time by the approaches mentioned above.

References

[1] R. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. Dongarra, *Matrix Market: a web resource for test matrix collections*, in R. Boisvert (ed.), *The Quality of Numerical Software: Assessment and Enhancement*, Chapman and Hall, London, 1997, `http://math.nist.gov/MatrixMarket/`.

[2] V. Chvátal, *A remark on a problem of Harary*, Czechoslovak Mathematics Journal **20** (1970), 95.

[3] G.M. Del Corso and G. Manzini, *Finding exact solutions to the bandwidth minimization problem*, Computing **62** (1999), 189–203.

[4] J.S. Turner, *On the probable performance of heuristics for bandwidth minimization*, SIAM Journal on Computing **15** (1986), 561–580.

[5] E.M. Gurari and I.H. Sudborough, *Improved dynamic programming algorithms for bandwidth minimization and the mincut linear arrangement problem*, Journal of Algorithms **5** (1984), 531–546.

Clustering Large Data Sets

Christian Sohler
(joint work with Gereon Frahling)

Clustering is the computational task to partition a given input into subsets of equal characteristics. These subsets are usually called clusters and ideally consist of similar objects that are dissimilar to objects in other clusters. This way one can use clusters as a coarse representation of the data. We loose the accuracy of the original data set but we achieve simplification (this is somewhat comparable to lossy compression). When we deal with large data sets clustering the data may be the only possibility to visualize the structure of the data set as visualizing the whole set is typically not possible.

Clustering has many other applications in different areas of computer sciences such as computational biology, machine learning, data mining and pattern recognition. Since the quality of a partition is rather problem dependent, there is no general clustering algorithm. In this talk we consider k-means clustering, which is a widely used formulation of clustering. In this problem we are given a set P of n points in the Euclidean space $\mathbb{R}^d$. The goal is to find a set C of k points (called *centers*), such that

$$\sum_{p \in P} (d(p,C))^2$$

is minimized, where $d(p,C)$ denotes the distance of point p to the nearest point in C.

In this talk, we want to develop a simple coreset construction for k-means clustering [1]. A coreset is a small weighted points sets (point weights stand for multiplicities of points) such that for any set of k centers the cost of the coreset is within $(1 \pm \epsilon)$ times the cost of the original point set. The coreset we compute has size in $O(\log n)$ for constant ϵ and d. We present a dynamic data structure (e.g., one supporting insertions and deletions) that maintains in $\text{poly}(\log n)$ space such a coreset for a sequence of n insertions and deletion of points. Once we have computed such a coreset we can use an arbitrary algorithm to obtain a good clustering.

In the second part of this talk we show how to use these coresets to obtain a fast implementation of Lloyd's algorithm [4], which is one of the most widely used heuristic for k-means clustering and clustering in general [2]. We start with a small coreset and run a variant of this algorithm on it. Then we move to a coreset of bigger size and run our variant on it using the solution from the previous coreset as starting solution. We continue this process until our coreset equals the whole point set. The variant of Lloyd's algorithm we use is a variant of the KMHybrid algorithm [3, 5].

References

[1] G. Frahling and C. Sohler, *Coresets in Dynamic Geometric Data Streams*, Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC'05), 2005, 209–217.

[2] G. Frahling and C. Sohler, *A fast k-means implementation using coresets*, Proceedings of the 22nd ACM Symposium on Computational Geometry (SoCG'06), 2006, to appear.

[3] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu, *An Efficient k-Means Clustering Algorithm: Analysis and Implementation*, IEEE Trans. Pattern Anal. Mach. Intell. **24**(7) (2002), 881-892.

[4] S. Lloyd, *Least Squares Quantization in PCM*, IEEE Transactions on Information Theory **28** (1982), 129–137.

[5] D. Mount, *A Testbed for k-Means Clustering Algorithms*, available at http://www.cs.umd.edu/mount/Projects/KMeans/km-local-doc.pdf.

Engineering B-trees and Cache-Oblivious B-trees on Real Memory Hierarchies (Ignorance is Bliss)

Michael A. Bender

(joint work with Martín Farach-Colton, Haodong Hu, and Bradley C. Kuszmaul)

In this talk we report on our experiences implementing B-trees and cache-oblivious B-trees. We then evaluate the predictive value of two memory models, the disk access machine (DAM) model, and the cache-oblivious (CO) model. The common perception is that CO algorithms, while elegant, loose a constant factor compared to DAM algorithms in order to pay for platform independence. However, the DAM does not model features of disks, such as prefetching of tracks, whereas the CO models data locality at all granularities and all levels of the hierarchy. The result is that for disk-bound data, the CO B-tree achieves comparable or superior performance even to a highly tuned B-tree.

References

[1] M. A. Bender and H. Hu, *An Adaptive Packed-Memory Array*, Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), 2006, 20–29.

[2] M. A. Bender, M. Farach-Colton, B. C. Kuszmaul, *Cache-Oblivious String B-Trees*, Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), 2006, 233–242.

Succinct Data Structures: A Survey

J. Ian Munro

Structural information is extremely useful, even crucial, to efficient information retrieval especially in its large scale forms such as data warehousing. However, the additional storage overhead of such information is often prohibitive. A recent approach to this problem has been the consideration of *succinct data structures*. The idea is to represent structural information, typicially a combinatorial object, in a number of bits close to the information theoretic lower bound for such a structure, but in a manner such that basic navigation opeations can be performed quickly, say in constant time.

Consider, for example a binary tree on n nodes. The natural represention requires left and right pointers, and probably parent pointers. Taking a pointer as $\lg n$ bits, the tree would require $3n \lg n$ bits to represent the tree, so that the obvious navigation operations of moving to either child or to the parent can be performed quickly. There are however, only $C_n = \binom{2n}{n}/(n+1)$ or about 4^n binary trees on n nodes, so about $2n$ bits suffice to identify such a tree. Jacobson seems to be the first to study such succinct representations upon which one can perform basic navigation operations, such as moving to a child or parent, quickly. He [1, 2] proposed an encoding of the binary tree, together with appended external nodes in all positions at which there are null pointers in the original tree. This gives a tree on $2n+1$ nodes. Internal nodes, i.e. the those in the original tree, are tagged with a 1, while the external nodes are tagged with a 0. The representation of the tree is the binary string of length $2n+1$ obtained by listing these tags from left to right and level by level through the tree. Navigation through the tree is perfomed with the aid of two auxiliary operations on a bit string: $rank(i)$ gives the number of 1's up to and including position i, while $select(i)$ gives the position of the i^{th} 1. Jacobson gave an index of $o(n)$ bits to support these operations in $O(\lg n)$ bit probes, which the author and others refined to a constant number of probes on a $\lg n$ bit RAM.

As the topic has developed, most of the work has focussed on succinct indices for full text search. Indeed the most natural structure for such searches is a suffix tree, which is essentially a binary tree with pointers to positions in the text at the leaves. The structure requires, in addition to the operations noted above, the ability to jump to an arbitrary leaf in a subtree and also tell subtree size. If, for example, the text string is over a 4 symbol alphabet, as is a genome, storing the index in the "naive" form requires about 100 times as much space as the raw text. Clearly this is not acceptable.

Other structures considered include binary trees on which updates are to be performed, ordered trees, planar graphs, permutations (with operations including find inverse) and arbitrary functions from $[n]$ to $[n]$. Munro and Rao ([3]) is a recent survey on the topic.

References

[1] G. Jacobson, *Space Efficient Static Trees and Graphs*, Proc. 30th IEEE Symp. on Foundations of Computer Science, 1989, 549–554.

[2] G. Jacobson, *Succinct Data Structures*, Technical Report CMU-CS-89-112, Carnegie Mellon University, 1989.

[3] J. I. Munro and S. S. Rao, *Succinct Representations of Data Structures*, Chapter 37 of *Handbook of Data Structures and Applications* (ed. D. P. Mehta and S. Sahni), Chapman & Hall/CRC, 2005.

Exact and Efficient Geometric Computing using Structural Filtering

Stefan Näher
(joint work with Martin Taphorn)

Geometric algorithms use geometric predicates in their conditionals. The common strategy for the exact implementation of geometric algorithms is to evaluate all geometric predicates exactly and to use floating point filters to make the exact evaluation of predicates fast. Floating-point filters have proved to be very efficient both in practice and in theory. The evaluation of a geometric predicate amounts to the computation of the sign of an arithmetic expression. A floating point filter evaluates the expression using floating point arithmetic and also computes an error bound to determine whether the floating point computation is reliable. If the error bound does not suffice to prove reliability, the expression is re-evaluated using exact arithmetic. Exact geometric computation incurs an overhead when compared to a pure floating point implementation. For “easy inputs” where the floating point computation always yields the correct sign, the overhead consists of the computation of the error bound. This overhead is about a factor of two for good filter implementations. For “difficult inputs” where the floating point filter always fails, the overhead may be much larger, but this is not really relevant, as the floating point computation will produce an incorrect result.

The challenge is to achieve exact geometric computation at the cost of floating point arithmetic. Structural filtering is a step in this direction. Structural filtering views the execution of an algorithm as a sequence of steps and applies filtering at the level of steps. A step may contain many predicate evaluations, errors are allowed in the evaluations of predicates, but the outcome of a step is guaranteed to be correct.

In this work we investigate the potential of structural filtering theoretically and experimentally. We give a classification of filtering techniques and compare our approach to filtering at the predicate and at the algorithm level. We show that predicate filtering is a special case of structural filtering and that structural filtering has the potential of improving upon predicate filtering for a wide class of algorithms.

The presented experimental results indicate that in many cases the possible speed-up of a factor of two can be achieved. This is in particular true for new implementations of algorithms for sorting, convex hull computations, plane sweep and point-location. Furthermore structural filtering can considerably improve the practical running time of range and segment trees.

References

[1] S. Funke, K. Mehlhorn, and S. Näher *Structural Filtering: a Paradigm for Efficient and Exact Geometric Programs*, Computational Geometry **31**(3) (2005), 179–194.

Improved External-Memory Breadth-First Search

Ulrich Meyer
(joint work with Deepak Ajwani and Vitaly Osipov)

We consider the problem of breadth first search (BFS) traversal on massive sparse undirected graphs in external memory. Engineering the algorithm of Munagala and Ranade [5] (MR_BFS) and the randomized and deterministic variants of the $o(n)$ I/O algorithm of Mehlhorn and Meyer [4] (MM_BFS_R and MM_BFS_D) coupled with a heuristic, we discuss the effect of various implementation design choices on the actual running time of the BFS traversal. Demonstrating the viability of our BFS implementations on various synthetic and real world benchmarks, we show that BFS level decompositions for large graphs (around a billion edges) can be computed on a cheap machine in a *few hours*. Concretely speaking, our contributions are the following (more details are provided in [1]):

- We improve upon the MR_BFS and MM_BFS_R implementation described in [2] by reducing the computational overhead associated with each BFS level, thereby improving the results for large diameter graphs.
- We discuss the various choices made for a fast MM_BFS_D implementation. This involved experimenting with various available external memory connected component and minimum spanning tree algorithms. Our partial re-implementation of the list ranking algorithm of [6] adapting it to the STXXL framework outperforms the other list ranking algorithms for the sizes of our interest. As for the Euler tour in the deterministic preprocessing, we compute the cyclic order of edges around the nodes using the STXXL sorting.
- We conduct a comparative study of MM_BFS_D with other external memory BFS algorithms and show that for most graph classes, MM_BFS_D outperforms MM_BFS_R. Also, we compare our BFS implementations with Christiani's implementations [3], which have some cache-oblivious subroutines. This gives us some idea of the loss factor that we will have to face for the performance of cache-oblivious BFS.
- We propose a heuristic for maintaining the pool in the BFS phase of MM_BFS. This heuristic improves the runtime of MM_BFS in practice, while preserving the worst case I/O bounds of MM_BFS.
- Putting everything together, we show that the BFS traversal can also be done on moderate and large diameter graphs in a few *hours*, which would have taken the implementations of [2] and [3] several *days* and internal-memory BFS several *months*. Also, on low diameter graphs, the time taken by our improved MR_BFS is around one-third of that in [2].

We summarize our results (Table 2) by giving the state of the art implementations of external memory BFS on different graph classes.

Graph class	n	m	MR_BFS	MM_BFS_R	MM_BFS_D
Random	2^{28}	2^{30}	**1.4**	7×	6×
Webgraph	$\sim 1.4 \cdot 10^8$	$\sim 1.2 \cdot 10^9$	**2.6**	3.5×	2×
Grid ($2^{14} \times 2^{14}$)	2^{28}	2^{29}	2.5×	1.25×	**21**
Grid ($2^{21} \times 2^7$)	2^{28}	$\sim 2^{29}$	>100×	>10×	**4.0**
Grid ($2^{27} \times 2$)	2^{28}	$\sim 2^{28} + 2^{27}$	>500×	>25×	**3.8**
Simple Line	2^{28}	$2^{28} - 1$	**0.4**	7×	7×
Random Line	2^{28}	$2^{28} - 1$	>1300×	>75×	**3.6**
Max			~ 1/2 **year**	~ 1 **week**	1 **day**

Table 2. The best total running time (in hours) for BFS traversal on different graphs with the best external memory BFS implementations; Entries like $> 25\times$ denote that this algorithm takes more than 25 times the time taken by the best algorithm for this input instance

References

[1] D. Ajwani, U. Meyer and V. Osipov *Improved external memory BFS implementations*, Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, 2007, to appear.

[2] D. Ajwani, R. Dementiev, and U. Meyer, *A computational study of external-memory BFS algorithms*, SODA, 2006, 601–610.

[3] Frederik Juul Christiani, *Cache-oblivious graph algorithms*, Master's thesis, Department of Mathematics and Computer Science, University of Southern Denmark, 2005.

[4] K. Mehlhorn and U. Meyer, *External-memory breadth-first search with sublinear I/O*, ESA, LNCS **2461**, 2002, 723–735.

[5] K. Munagala and A. Ranade, *I/O-complexity of graph algorithms*, SODA, 1999, ACM-SIAM, 687–694.

[6] J. F. Sibeyn, *From parallel to external list ranking*, Technical report, Max Planck Institut für Informatik, Saarbrücken, Germany, 1997.

Significance-Driven Graph Clustering

Dorothea Wagner
(joint work with Marco Gaertler and Robert Görke)

Modularity, the recently defined quality measure for clusterings, has attained instant popularity in the fields of social and natural sciences. We revisit the rationale behind the definition of modularity and *explore* the founding paradigm. This paradigm is based on the trade-off between the achieved quality and the expected quality of a clustering with respect to networks with similar intrinsic structure. We experimentally evaluate realizations of this paradigm systematically, including modularity, *and describe efficient algorithms for their optimization.* We confirm the feasibility of the resulting generality by a first systematic analysis of the behavior of these realizations on both artificial and on real-world data, arriving at remarkably good results of community detection.

This paper will appear in *Proceedings of 3rd International Conference on Algorithmic Aspects in Information and Management (AAIM'2007), Springer LNCS.*

External-Memory and Cache-Oblivious Algorithms: Theory and Experiments

Gerth Stølting Brodal

Modern computers get more and more complicated memory hierarchies. A typical computer contains a CPU with a limited number registers, several layers of caching (L1, L2 and L3), main memory, and secondary storage on disk. The layers are characterized by having different sizes, access times and block sizes. Traditional algorithm design ignores these factors when designing algorithms, even that these parameters can have significant impact on the performance of an algorithm - e.g. the performance of HeapSort is reduced by several orders of magnitude when data exceeds main memory size whereas MergeSort does not have this problem. A lot of experimental work has recently studied these influences on basic algorithmic problems, and many algorithms have been developed with the aim of reducing the number of disk I/Os, cache-faults, TLB misses etc.

In the talk examples were presented which show how the hardware parameters from the memory hierarchy affect the running time of simple algorithms, including the influence of TLB misses, prefetching of data, and branch prediction. In particular we showed experimental results for algorithms designed for the abstract external-memory model by Aggarwal and Vitter [1] and cache-oblivious model by Frigo *et al.* [5], which in a simplified manor try to take the memory hierarchies into account. Experimental results for cache-oblivious algorithms were presented for sorting [3], search trees [2], and matrix multiplication [5]. Furthermore experimental examples were presented for the influence of TLB misses on the performance of Radix Sort [8], the influence of L2 prefetching versus the cost of cache-faults for shortest path algorithms [7], and the cost of branch mispredictions versus the cost of caching for skewed versions of Quicksort [6] and skewed search trees [4].

References

[1] A. Aggarwal and J. S. Vitter, *The input/output complexity of sorting and related problems*, Communications of the ACM, **31**(9) (1988), 1116–1127.

[2] G. S. Brodal, R. Fagerberg, and R. Jacob, *Cache-oblivious search trees via binary trees of small height*, Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms, 2002, 39–48.

[3] G. S. Brodal, R. Fagerberg, and K. Vinther, *Engineering a cache-oblivious sorting algorithm*, ACM Journal of Experimental Algorithmics, Special Issue of ALENEX 2004, **12**, 2007, 23, Article No. 2.2.

[4] G. S. Brodal and G. Moruz, *Skewed binary search trees*, Proc. 14th Annual European Symposium, LNCS **4168** (2006), 708–719.

[5] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, *Cache-oblivious algorithms*, Proc. 40th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1999, 285–297.
[6] K. Kaligosi and P. Sanders, *How branch mispredictions affect quicksort*, Proc. 14th Annual European Symposium, LNCS **4168** (2006), 780–791.
[7] S. Pan, C. Cherng, K. Dick, and R. E. Ladner, *Algorithms to take advantage of hardware prefetching*, Proc. 9th Workshop on Algorithms Engineering and Experimentation (ALENEX '07), 2007.
[8] N. Rahman and R. Raman, *Adapting radix sort to the memory hierarchy*, J. Exp. Algorithmics **6** (2001), 7.

Graph Clustering based on Disturbed Diffusion

Henning Meyerhenke
(joint work with Burkhard Monien, Stefan Schamberger, and Thomas Sauerwald)

Graph clustering refers to the placement of nodes into meaningful groups based on some similarity measure. It is an important task in a wide variety of applications, e. g., network analysis for community detection. We address this in general $\mathcal{NP}$-hard problem by a heuristic algorithm we also use for partitioning graphs arising in parallel adaptive numerical simulations [1]. Its iterative approach resembles Lloyd's k-means algorithm [2], but its similarity measure is derived from a disturbed diffusion scheme called FOS/C instead of geometric distances.

Definition 1. [1] *Given an edge-weighted graph $G = (V, E, \omega)$ with n nodes and m edges, a set of source nodes S, and constants $0 < \alpha \leq (\text{degree}(G) + 1)^{-1}$ and $\delta > 0$.*[1] *Let the initial load vector $w^{(0)}$ and the drain vector d (which is responsible for the disturbance) be defined as follows:*

$$w_v^{(0)} = \begin{cases} \frac{n}{|S|} & v \in S \\ 0 & \textit{otherwise} \end{cases} \qquad d_v = \begin{cases} \frac{\delta n}{|S|} - \delta & v \in S \\ -\delta & \textit{otherwise} \end{cases}$$

Then, the FOS/C diffusion scheme performs the following operations in each iteration:

$$f_{e=(u,v)}^{(t)} = \alpha(w_u^{(t)} - w_v^{(t)}),\ w_v^{(t+1)} = (w_v^{(t)} + \sum_{e=\{*,v\}} f_e^{(t)}) + d_v.$$

By using FOS/C for measuring distances between nodes within Lloyd's algorithm, our heuristic focuses on good cluster or partition shapes rather than on minimizing a classical cut metric. This approach contrasts to most existing graph partitioning libraries [3, 4, 5]. For graph clustering, however, there are a number of related graph clustering heuristics [6, 7, 8]. Yet, they have at least one disadvantage with respect to efficiency, quality, parameter determination, or theoretical properties.

In practice the convergence state of an FOS/C system is determined by solving a linear system with sparse iterative solvers such as conjugate gradient or

[1] Here, the maximum degree is defined as $\text{degree}(G) := \max_{u \in V} \sum_{e=\{u,v\} \in E} \omega(e)$.

algebraic multigrid. This is reasonably efficient (typically $\mathcal{O}(n^{3/2})$ or better) and the combination of Lloyd's algorithm with FOS/C is shown to converge towards a local optimum of a potential function if no restriction is made on the cluster sizes [9]. This convergence happens very quickly when a multilevel scheme known from graph partitioning [10] is used.

Our experiments reveal promising results. The clustering quality on random and real-world data sets is mostly really good [9], but difficult instances require further work such as fine-tuned local search. Regarding graph partitioning, our algorithm computes high-quality partitions: they are connected, have a low diameter and few boundary vertices [1]. The quality compares very often favorably to the other state-of-the-art graph partitioning libraries cited above regarding these metrics, while the edge-cut is comparable or only slightly increased. We also hint at some speedup techniques [11] in order to overcome the speed difference to these competitors.

Finally, we present a proof-of-concept to use the above ideas for partitioning dynamic graphs with local knowledge only, i. e., when vertices are only allowed to access information stored at themselves and their neighbors.

References

[1] H. Meyerhenke, B. Monien, and S. Schamberger, *Accelerating shape optimizing load balancing for parallel FEM simulations by algebraic multigrid*, Proc. 20th IEEE Intl. Parallel & Distrib. Proc. Symp. (IPDPS), 2006, 57 (CD).

[2] S. P. Lloyd, *Least squares quantization in PCM*, IEEE Trans. on Information Theory, **28**(2) (1982), 129–136.

[3] G. Karypis and V. Kumar, *MeTis: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, [...], Version 4.0*, 1998.

[4] S. Schamberger, *Graph partitioning with the Party library: Helpful-sets in practice*, Comp. Arch. and High Perf. Comp., 2004, 198–205.

[5] C. Walshaw, *The parallel JOSTLE library user guide: Version 3.0*, 2002.

[6] S. Lafon and A. Lee, *Diffusion maps and coarse-graining: A unified framework for dimensionality reduction, graph partioning and data set parametrization*, IEEE Trans. on Pattern Analysis and Machine Intelligence, **28**(9) (2006), 1393–1403.

[7] I. S. Dhillon, Y. Guan, and B. Kulis, *Weighted graph cuts without eigenvectors: A multilevel approach*, IEEE Trans. on Pattern Analysis and Machine Intelligence, to appear.

[8] L. Yen, D. Vanvyve, F. Wouters, F. Fouss, M. Verleysen, and M. Saerens, *Clustering using a random-walk based distance measure*, Proc. 13th European Symposium on Artificial Neural Networks (ESANN), 2005, 317–324.

[9] H. Meyerhenke, B. Monien, T. Sauerwald, *Efficient k-means-type Clustering of Sparse Graphs with Provable Convergence*, 2007, submitted.

[10] B. Hendrickson and R. Leland, *A multi-level algorithm for partitioning graphs*, Supercomputing'95, 1995.

[11] H. Meyerhenke and S. Schamberger, *A parallel shape optimizing load balancer*, Proc. Euro-Par 2006, 2006, 232–242.

Resilient priority queues

Gabriel Moruz
(joint work with Allan G. Jørgensen and Thomas Mølhave)

Abstract. In the faulty-memory RAM model, the content of memory cells can get corrupted at any time during the execution of an algorithm, and a constant number of uncorruptible registers are available. A resilient data structure in this model works correctly on the set of uncorrupted values. In this paper we introduce a resilient priority queue. The deletemin operation of a resilient priority queue returns either the minimum uncorrupted element or some corrupted element. Our resilient priority queue uses $O(n)$ space to store n elements. Both insert and deletemin operations are performed in $O(\log n + \delta)$ time amortized, where δ is the maximum amount of corruptions tolerated. Our priority queue matches the performance of classical optimal priority queues in the RAM model when the number of corruptions tolerated is $O(\log n)$. We prove matching worst case lower bounds for resilient priority queues storing only structural information in the uncorruptible registers between operations.

Motivation. Memory devices continually become smaller, work at higher frequencies and lower voltages, and in general have increased circuit complexity [2]. Unfortunately, these improvements come at the cost of reliability [8, 6]. A number of factors, such as alpha particles, infrared radiation, and cosmic rays, can cause *soft memory errors* where a bit flips and as a consequence the value stored in the corresponding memory cell is corrupted. Many modern computing centers consist of relatively cheap of-the-shelf components, and the large number of individual memories involved in these clusters substantially increase the frequency of memory corruptions in the system. Hence it is crucial that the software running on these machines is robust. Since the amount of cosmic rays increases dramatically with altitude, soft memory errors are of special concern in fields like avionics or space research. Furthermore, soft memory error rates are expected to rise for both DRAM and SRAM memories [8].

Traditionally, the work within the algorithmic community has focused on models where the integrity of the memory system is not an issue. In these models, the corruption of even a single memory cell can have a dramatic effect on the output. For instance, a single corrupted value can induce as much as $\Theta(n^2)$ inversions in the output of a standard implementation of mergesort [3]. Replication can help in dealing with corruptions, but is not always feasible, since the time and space overheads are not negligible.

Faulty-memory RAM.. Finocchi and Italiano [3] introduced the *faulty-memory random access machine*, which is a random access machine where the content of memory cells can get corrupted at *any time* and at *any location.* Corrupted cells cannot be distinguished from uncorrupted cells. The model is parametrized by an upper bound δ on the number of corruptions occurring during the lifetime of an algorithm. It is assumed that $O(1)$ reliable memory cells are provided, a reasonable assumption since CPU registers are considered reliable. Also, copying an element is considered an atomic operation, *i.e.* the elements are not corrupted while being copied. An algorithm is *resilient* if it is able to achieve a correct output at least for the uncorrupted values. This is the best one can hope for, since the output can get corrupted just after the algorithm finishes its execution. For instance a resilient sorting algorithm guarantees that there are no inversions between the uncorrupted elements in the output sequence.

Several important results have been achieved in the faulty-memory RAM. In the original paper, Finocchi and Italiano [3] proved lower bounds and gave (non-optimal) resilient algorithms for sorting and searching. Algorithms matching the lower bounds for sorting and searching(expected time) were presented in [4]. An optimal resilient sorting algorithm takes $\Theta(n \log n + \delta^2)$ time, whereas optimal searching is performed in $\Theta(\log n + \delta)$ time. Furthermore, in [5] a resilient search tree that performs searches and updates in $O(\log n + \delta^2)$ time amortized was developed. Finally, in [7] it was shown that resilient sorting algorithms are of practical interest.

Our contributions. In this paper we design and analyze a priority queue in the faulty-memory RAM model. A resilient priority queue is a data structure that maintains a set of elements under the operations INSERT and DELETEMIN as follows. An INSERT adds an element and a DELETEMIN deletes and returns the minimum uncorrupted element or a corrupted one.

Our priority queue uses $O(n)$ space for storing n elements and supports both INSERT and DELETEMIN in $O(\log n + \delta)$ time amortized. It matches the bounds for an optimal comparison based priority queue in the RAM model while tolerating $O(\log n)$ corruptions. It is a significant improvement over using the resilient search tree in [5] as a priority queue, since it uses $O(\log n + \delta^2)$ time amortized per operation and thus only tolerates $O(\sqrt{\log n})$ corruptions to preserve the $O(\log n)$ bound per operation. Our priority queue is the first resilient data structure allowing $O(\log n)$ corruptions, while still matching optimal bounds in the RAM model. Our priority queue does not store elements in reliable memory between operations, only structural information like pointers and indices. We prove that any comparison based resilient priority queue behaving this way requires worst case $\Omega(\log n + \delta)$ time for either INSERT or DELETEMIN.

The resilient priority queue is based on the cache-oblivious priority queue by Arge *et al.* [1]. The main idea is to gather elements in large sorted groups of increasing size, such that expensive updates do not occur too often. The smaller groups contain the smaller elements, so they can be retrieved faster by DELETEMIN operations. We extensively use the resilient merging algorithm in [4] to move elements among the groups. Due to the large sizes of the groups, the extra work required to deal with corruptions in the merging algorithm becomes insignificant compared to the actual work done.

REFERENCES

[1] L. Arge, M.A. Bender, E.D. Demaine, B. Holland-Minkley, J.I. Munro, *Cache-oblivious priority queue and graph algorithm applications*, Proc. 34th Annual ACM Symposium on Theory of Computing, 2002, 268–276.

[2] C. Constantinescu, *Trends and challenges in VLSI circuit reliability*, IEEE micro **23**(4) (2003), 14–19.
[3] I. Finocchi, G.F. Italiano, *Sorting and searching in the presence of memory faults (without redundancy)*, Proc. 36th Annual ACM Symposium on Theory of Computing, 2004, 101–110.
[4] I. Finocchi, F. Grandoni, G.F. Italiano, *Optimal resilient sorting and searching in the presence of memory faults*, Proc. 33rd Int. Colloquium on Automata, Languages and Programming, 2006, 286–298.
[5] I. Finocchi, F. Grandoni, G.F. Italiano, *Resilient search trees*, Proc. 18th ACM-SIAM Symposium on Discrete Algorithms, 2007, to appear.
[6] A.J. van de Goor, *Testing Semiconductor Memories: Theory and Practice*, ComTex Publishing, Gouda, The Netherlands, 1998, ISBN 90-804276-1-6.
[7] U.F. Petrillo, I. Finocchi, G.F. Italiano, *The price of resiliency: a case study on sorting with memory faults*, Proc. 14th Annual European Symposium on Algorithms, 2006, 768–779.
[8] Tezzaron Semiconductor, *Soft errors in electronic memory - a white paper*, http://www.tezzaron.com/about/papers/papers.html, 2004.

A Platform for Engineering Cache-Oblivious Algorithms and Data Structures: The Architecture

Luca Allulli
(joint work with Fabrizio d'Amore and Enrico Puddu)

Cache-oblivious algorithms [6] are designed to be executed on the *ideal-cache* machine, an abstract machine which faithfully models real-world machines with hierarchical memory. Algorithms for the ideal-cache machine, as well as algorithm for the RAM, work with a semi-infinite memory space. But while in the uniform RAM model it is assumed that every memory access has the same cost, which is incorrect for large data sets, the semi-infinite memory of the ideal-cache machine models the virtual memory space of a computer, and block transfers to a faster memory level are taken into account by the model. Nevertheless, implementing cache-oblivious algorithms and data structures is not an easy task, because they use peculiar techniques that are not (easily) supported by general purpose programming languages and environments such as C, C++, or Java. For example, consider the following issues, that arise when implementing a cache-oblivious algorithm:

- **Memory management**. General-purpose memory managers, such as common implementations of the `malloc()` function, keep linked lists of the memory areas containing deallocated memory elements. When a new memory element is allocated, the memory manager uses a linked list to find a free large enough memory area. If the element pointed by the linked list is not in cache, the allocation provokes one cache miss.
- **Usage of pointers to perform elaborated tasks**. In C++ only simple tasks can be performed without using pointers. Allocating an array whose size is not known at compile time, or creating a record-like structure which "contains" an object whose size is not known at compile time, is normally achieved through pointers. Pointers are problematic for cache-oblivious algorithms and data structures, because following a pointer may cause

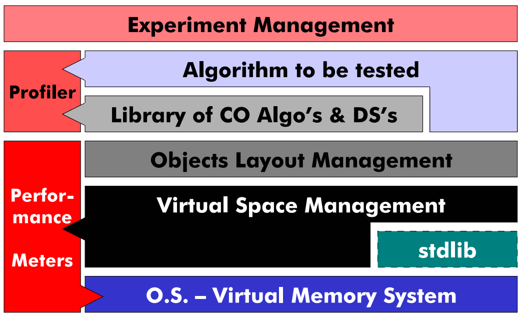

FIGURE 1. Architecture of the platform

a cache miss. The solution is to build complex memory layouts, often described by recursive definitions; but these layouts are difficult to obtain using standard C++ only.

To find a solution to the mentioned issues, and since a library of cache-oblivious algorithms and data structures did not exist, we decided to create a platform for engineering cache-oblivious algorithms and data structures. The purpose of our platform is twofold: to facilitate the implementation of algorithms, so that they should be usable in real-world applications; and to support the engineering through experimental analysis of such algorithms. The architecture of our platform is depicted in Fig. 1.

Operating System - Virtual Memory System. We consider UNIX-like operating systems, in particular Linux. If we ignore memory areas used by the run-time system (such as the function stack, the executable code etc.), a process can *reserve* the areas of its memory space it wants to use (so that they get paged) using two system functions: the `brk()` function, by which the process sets the limit of the prefix of its memory space it wants to use; and the `mmap()` function, which allows it to reserve arbitrary areas of the memory space.

Memory Management Layer. The memory management layer hosts several memory managers. A *memory manager* is an object that provides the following functions: `Allocate(size_t size)`, which allocates a new memory element of size `size`; `Deallocate(long memel)` and `Resize(long memel, size_t newsize)`, respectively deallocating and resizing a memory element. The reason why several memory managers are provided is that each manager is characterized by a peculiar allocation/deallocation policy, and by a way of requesting memory.

Memory can be requested directly to the operating system via the `brk()` function or the `mmap()` function: in this way, we can logically separate the algorithms and data structures which compose the cache-oblivious algorithm to be engineered, and separately analyze the memory usage (including cache misses) of each component. Alternatively, a memory manager may use any implementation of the C malloc() function to get memory. Finally, a memory manager may allocate the memory it needs from another memory manager.

We now exemplify some of the memory allocation policies. One of these is a **stack-like** policy: the memory manager "simulates" the function stack, in the sense that memory elements are allocated in an "activation record", and deallocated as soon as the function in which they were allocated terminates. In this way, algorithms can easily allocate on a stack arrays whose size is known only at run time. With a **non-deallocating** policy the manager never tries to reuse deallocated memory space: this solution does not increase the asymptotic I/O cost of algorithms, and may be adopted to "simulate" preallocation, using a constant factor of the time, space and I/O's that would be used by preallocation. In order to save space, **compacting** memory managers can be used, which recover unused reserved space when a constant fraction of the reserved space has been deallocated. The I/O cost of moving data in memory is amortized on the cost of accessing every allocated element at least once; but there is an additional cost, for updating pointers, that in general cannot be amortized. Notice, however, that even internal-memory allocators in general cause additional costs that are not considered in the theoretical analysis of internal memory algorithms.

Layout Management Layer. In order to create complex memory layouts, the programmer should define *enhanced classes* that, differently from ordinary classes, may have a size determined at run time. For each enhanced class C, a *factory class* is defined; its *factory objects* are in charge of creating objects of class C. A factory object is fed with information known only at run time, such as the factory objects of the sub-objects of the enhanced object to be created, the size of arrays, etc.; with this information it computes the size of the enhanced object, asks for space to a memory manager, creates the enhanced object, and invokes a build function on the factory objects of its sub-objects.

Library of Cache-Oblivious Algorithms and Data Structures; Target Algorithm. The first motivation of our platform was an effort to engineer the cache-oblivious single source shortest paths algorithms of Allulli *et al.* [1]. In order to implement it, we developed a library of cache-oblivious algorithms and data structures that currently includes tools for scanning and sorting, median selection [6], a priority queue [2], tools for the time-forward processing [5], list ranking [5], the minimum spanning forest [3], and the Euler tour computation.

Performance Meters, Profiler, and Experiment Manager. Our platform can be compiled in two modes: a *simulated* mode, where every memory access is tracked, and a *real* mode, where this does not happen. In the former it is possible to simulate an arbitrary memory hierarchy. In the latter it is still possible to get several useful cost measures from the system, such as time, number of page faults etc. An optional profiler allows to attribute each measured cost to the function that generated it. An experiment manager runs experiments in batch.

Status. All the mentioned layers, algorithms and data structures have been implemented. As a sorting algorithm, we currently use Mergesort. Before releasing our code we need to (1) improve the generality of a portion of the library w.r.t. the memory management layer; (2) implement Funnelsort [6], or integrate the implementation of Brodal *et al.* [4] into our platform; and (3) perform a first engineering step in order to improve memory usage.

References

[1] L. Allulli, P. Lichodzijewski, and N. Zeh, *A faster cache-oblivious shortest-path algorithm for undirected graphs with bounded edge lengths*, SODA07, 2007, Society for Industrial and Applied Mathematics.

[2] L. Arge, M.A. Bender, E.D., Demaine, B. Holland-Minkley, and J.I. Munro, *Cache-oblivious priority queue and graph algorithm applications*, STOC02, 2002, ACM Press, 268–276.

[3] L. Arge, G.S. Brodal, and L. Toma, *On external-memory MST, SSSP and multi-way planar graph separation*, J. Algorithms **53**(2) (2004), 186–206.

[4] G.S. Brodal, R. Fagerberg, and K. Vinther, *Engineering a cache-oblivious sorting algorithm*, ACM J. of Experimental Algorithmics **12**(2.2), 2007.

[5] Y.J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, and J.S. Vitter, *External-memory graph algorithms*, SODA95, 1995, Society for Industrial and Applied Mathematics, 139–149.

[6] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran, *Cache-oblivious algorithms*, FOCS99, 1999, IEEE Computer Society Press, 285–297.

Sequential Vector Packing

Riko Jacob

(joint work with Mark Cieliebak, Alexander Hall, and Marc Nunkesser)

We introduce a novel variant of the well known d-dimensional bin (or vector) packing problem. Given a sequence of non-negative d-dimensional vectors, the goal is to pack these into as few bins as possible of smallest possible size. In the classical problem the bin size vector is given and the sequence can be partitioned arbitrarily. We study a variation where the vectors have to be packed in the order in which they arrive and the bin size vector can be chosen once in the beginning. This setting gives rise to two combinatorial problems: One in which we want to minimize the number of used bins for a given total bin size and one in which we want to minimize the total bin size for a given number of bins. We prove that both problems are $\mathcal{NP}$-hard and propose an LP based bicriteria $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$-approximation algorithm. We give a 2-approximation algorithm for the version with bounded number of bins. Furthermore, we investigate properties of natural greedy algorithms, and present an easy to implement heuristic, which is fast and performs well in practice.

This work was presented at the ESCAPE conference in 2007, and appears in the proceedings published in the LNCS series of Springer.

Reporters: Carsten Gutwenger and Joachim Reichel

Participants

Prof. Dr. Susanne Albers
Institut fuer Informatik
Lehrstuhl Paralleles und Verteiltes
Rechnen, Albert-Ludwigs-Universität
Georges-Köhler-Allee 79
79110 Freiburg

Luca Allulli
Dipartimento di Scienze
dell' Informazione
Universita di Roma "La Sapienza"
Via Salaria 113
I-00198 Roma

Prof. Dr. Michael Bender
Department of Computer Science
State University of New York at
Stony Brook
Stony Brook , NY 11794-4400
USA

Vincenzo Bonifaci
Dipartimento di Informatica e
Sistemistica
Universita di Roma "La Sapienza"
Via Salaria 113
I-00198 Roma

Prof. Dr. Gerth Brodal
Dept. of Computer Science
University of Aarhus
IT-Parken, Aabogade 34
DK-8200 Aarhus N

Prof. Dr. Alberto Caprara
DEIS
Universita di Bologna
Viale Risorgimento, 2
I-40136 Bologna

Markus Chimani
Fachbereich Informatik LS11
Universität Dortmund
Otto-Hahn-Straße 14
44227 Dortmund

Prof. Dr. William J. Cook
Industrial and Systems Engineering
Georgia Institute of Technology
Atlanta , GA 30332-0205
USA

Dr. Camil Demetrescu
Dipartimento di Informatica e
Sistemistica
Universita di Roma "La Sapienza"
Via Salaria 113
I-00198 Roma

Dr. Andreas Döring
Institut für Informatik
Freie Universität Berlin
Takustr. 9
14195 Berlin

Prof. Dr. Friedrich Eisenbrand
Institut für Mathematik
Universität Paderborn
33095 Paderborn

Prof. Dr. Rolf Fagerberg
Department of Mathematics and
Computer Science
University of Southern Denmark
Campusvej 55
DK-5230 Odense M

Prof. Dr. Irene Finocchi
Dept. of Computer Science
Universita "La Sapienza"
Via Salaria 113
I-00198 Roma

Leonor Frias Moya
Dept. de Llenguatges i Sist. Informatics
Universitat Politecnica de Catalunya
Campus Nord-Edifici, Omega, s109
Jordi Girona Salgado, 1-3
E-08034 Barcelona

Carsten Gutwenger
Fachbereich Informatik LS11
Universität Dortmund
Otto-Hahn-Straße 14
44227 Dortmund

Prof. Dr. Giuseppe F. Italiano
Dipartimento di Informatica,
Sistemi e Produzione
Universita di Roma "Tor Vergata"
via del Politecnico 1
I-00133 Roma

Riko Jacob
Institut für theoretische
Informatik
ETH-Zentrum
Universitätstr.6
CH-8092 Zürich

Prof. Dr. Michael Jünger
Institut für Informatik
Universität zu Köln
Pohligstr. 1
50969 Köln

Prof. Dr. Jyrki Katajainen
Dept. of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen

Dr. Thorsten Koch
Konrad-Zuse-Zentrum für
Informationstechnik Berlin (ZIB)
Takustr. 7
14195 Berlin

Dr. Luigi Laura
Dipartimento di Scienze
dell' Informazione
Universita di Roma "La Sapienza"
Via Salaria 113
I-00198 Roma

Prof. Dr. Friedhelm Meyer auf der Heide
Heinz-Nixdorf Institut &
Institut für Informatik
Universität Paderborn
Fürstenallee 11
33102 Paderborn

Dipl.Inf. Henning Meyerhenke
Fakultät EIM - Elektrotechnik,
Informatik und Mathematik
Universität Paderborn
Fürstenallee 11
33102 Paderborn

Prof. Dr. Ulrich C. Meyer
Fachbereich Mathematik/Informatik
J.W.Goethe-Universität
Robert-Mayer-Str. 11-15
60325 Frankfurt/M.

Prof. Dr. Rolf Möhring
Institut für Mathematik - Fak. II
Technische Universität Berlin
Sekr. MA 6-1
Straße des 17. Juni 136
10623 Berlin

Prof. Dr. Burkhard Monien
Heinz-Nixdorf Institut &
Institut für Informatik
Universität Paderborn
Fürstenallee 11
33102 Paderborn

Gabriel Moruz
BRICS, Dept. of Computer Science
University of Aarhus
Ny Munkegade, Building 540
DK-8000 Aarhus C

Prof. Dr. Ian Munro
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
CANADA

Prof. Dr. Petra Mutzel
Fachbereich Informatik LS11
Universität Dortmund
Otto-Hahn-Straße 14
44227 Dortmund

Prof. Dr. Stefan Näher
Abteilung Informatik
Fachbereich IV
Universität Trier
54286 Trier

Prof. Dr. Ulrich Pferschy
Institut für Statistik und
Operations Research
Universität Graz
Universitätsstraße 15
A-8010 Graz

Prof. Dr. Tomasz Radzik
Department of Computer Science
King's College London
Strand
GB-London WC2R 2LS

Prof. Dr. Rajeev Raman
Department of Computer Science
University of Leicester
University Road
GB-Leicester LE1 7RH

Dr. Joachim Reichel
Fachbereich Mathematik
Universität Dortmund
44221 Dortmund

Prof. Dr. Gerhard Reinelt
Institut für Informatik
Ruprecht-Karls-Universität
Heidelberg
Im Neuenheimer Feld 368
69120 Heidelberg

Prof. Dr. Giovanni Rinaldi
Istituto di Analisi dei Sistemi ed
Informatica
CNR
Viale Manzoni 30
I-00185 Roma

Heiko Röglin
Lehrstuhl für Informatik I
RWTH Aachen
Ahornstr. 55
52074 Aachen

Prof. Dr. Peter Sanders
Universität Karlsruhe
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Dominik Schultes
Institut für Informatik
Universität Karlsruhe
76128 Karlsruhe

Johannes Singler
Institut für Informatik
Universität Karlsruhe
76128 Karlsruhe

Prof. Dr. Martin Skutella
Fachbereich Mathematik
Universität Dortmund
44221 Dortmund

Prof. Dr. Christian Sohler
Heinz-Nixdorf Institut &
Institut für Informatik
Universität Paderborn
Fürstenallee 11
33102 Paderborn

Prof. Dr. Paul Spirakis
University of Patras & (RA) CTI
Dept. of Comp. Eng. and Informatics
University Campus
N. Kazantzakis str.
26500 Rio, Patras
GREECE

Prof. Dr. Dorothea Wagner
Universität Karlsruhe
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Prof. Dr. Ingo Wegener
Lehrstuhl für Informatik II
Universität Dortmund
44221 Dortmund

Prof. Dr. Christos Zaroliagis
University of Patras & (RA) CTI
Dept. of Comp. Eng. and Informatics
University Campus
N. Kazantzakis str.
26500 Rio, Patras
GREECE

Prof. Dr. Norbert Zeh
Faculty of Computer Science
Dalhousie University
6050 University Avenue
Halifax , N.S. B3H 1W5
CANADA