

## 1) Research overview

This document provides a brief overview of my research activities until June 2006, based on 10 selected publications. To keep the text short, some simplifications have been made (you don't get the full picture), and there is almost no comparison to related work – for more information consult the papers. The document also outlines future research directions. The text is intended to be accessible (at least) for people with a degree in computer science, or similar.

### Background

Many computer applications require the manipulation or storage of a *set* of data elements. Most of my research has revolved around algorithms and data structures for sets. The *practical* goal of research in algorithms and data structures is to devise methods for performing data manipulation and storage tasks, that are *efficient* with respect to the use of computing resources, and *scalable* in the sense that the methods remain efficient as the amount of data grows. The quest towards this goal has two main ingredients:

- A theoretical understanding of the algorithmic limits and possibilities.
- Algorithm engineering techniques for transformation of algorithms into efficient software.

In a sense, a recurring theme in much of my work is providing a bridge from the former to the latter of the two ingredients. All too often, the algorithms known to be best for large data sets are hopelessly complicated to implement, or less efficient than other alternatives when run on the data encountered in practice. In my work I emphasize simple and easy-to-implement, yet theoretically well-understood methods. As I will argue below (and as history has shown), the theoretical understanding is in fact *instrumental* in designing good algorithms — not something that is just added “after the fact.” Needless to say, the goal of designing algorithms that are both practically and theoretically good has not been reached in all cases. Indeed, this is the reason to continue!

### A tale of ten papers

Many algorithms for handling sets are *randomized*, meaning that the behavior of the algorithm is in part determined by random choices. The analysis of such algorithms typically shows that they perform well in the expected sense, or with high probability. This leaves the problem of how to perform (truly) random choices in a computer, as well as the problem that the algorithm may fail to deliver the expected performance. The problem of storing a set of keys such that any key can be efficiently found in memory is called the *dictionary* problem. It is one of the most prominent, and practically important data structuring problems. However, the currently fastest dictionary data structures are randomized. In the paper

*Faster Deterministic Dictionaries (proceedings of SODA 2000)*

I present a deterministic (i.e., non-randomized) algorithm that constructs a dictionary with constant lookup time in  $O(n \log n)$  time, where  $n$  is the size of the set. This remains the fastest known deterministic algorithm for constructing dictionaries with constant lookup time. Further possibilities for design of deterministic dictionaries were recently explored in

*Deterministic load balancing and dictionaries in the parallel disk model (proceedings of SPAA 2006)*

where we consider the *dynamic* dictionary problem in which insertions/deletions into/from the set is possible. We show that if sufficient *parallelism* is available (as e.g. in an array of disks) the performance of known randomized dictionaries can, in theory, be matched deterministically.

Motivated by large data sets, as well as small-memory devices, the space usage of dictionaries is an important performance parameter. In the paper

*Low Redundancy in Static Dictionaries with Constant Query Time (SIAM J. on Computing, 2001)*

I present a dictionary with space usage  $B + o(n)$  bits, where  $B$  is the best space usage one could possibly hope for. This is the currently smallest space usage for a dictionary with constant lookup time. A systematic study of the theoretical possibilities and limits of dictionaries was presented in

*On the Cell Probe Complexity of Membership and Perfect Hashing (proceedings of STOC 2001).*

One of the main findings was that the fastest dictionary one can hope for is one that (in addition to some initial computation) looks up a key by looking independently at two memory locations. It was shown that it is possible to arrange the keys in memory such that any key in the set can be found in this way. In the paper

*Cuckoo Hashing (Journal of Algorithms, 2004)*

we show how to realize this in a dynamic setting, using an extremely simple procedure for updating the data structure. It is unlikely that this result would have come about in a purely experimental setting: Even though the insertion procedure is a natural one to try, it only works if the space usage of the algorithm is slightly larger than usual in a hash table. Thus, experimental work would likely conclude that the algorithm does not work.

It is a fact that many dictionary implementations based on hashing in use today are *heuristics*, and are either not theoretically understood, or provably have bad behavior in certain situations. Many of these methods have been analyzed under the (false) assumption of ideal *uniform* hash functions. In the paper

*Uniform Hashing in Constant Time and Linear Space (proceedings of STOC 2003)*

we show that it is possible to *simulate* the ideal of uniform hashing with constant time and space overhead. This provides a theoretical bridge between what can be analyzed in mathematical terms and the commonly used heuristics.

Not all applications of dictionaries need exact answers. Indeed, there is a number of applications in which it is tolerable that some key not in the set seems to be in the set. The idea of considering this relaxed version of the dictionary problem is that the space usage of the data structure can be reduced considerably, as observed by Bloom around 1970. Recently, there has been renewed interest in Bloom's solution to this problem ("Bloom filters"). In the paper

*An Optimal Bloom Filter Replacement (proceedings of SODA 2005)*

we present an alternative approach, resulting in a data structure that, at least in theory, improves upon Bloom's (and its successors) in a number of performance criteria. In fact, this data structure is, in an asymptotic sense, optimal.

When a set of keys is ordered it makes sense to search for keys in an interval  $[a; b]$  (a *range query*). Indeed, this is a very common type of search in database systems. In the paper

*On Dynamic Range Reporting in One Dimension (proceedings of STOC 2005)*

we show how to combine fast updates with range queries that are exponentially faster than previously known, in a linear space data structure.

Sorting a set of keys is an important and extremely well-studied problem. Most operations performed in database systems include sorting of data on external memory (i.e., disk) as a crucial ingredient. In the paper

*External String Sorting: Faster and Cache-Oblivious (proceedings of STACS 2006)*

we present the currently asymptotically fastest algorithm for sorting a set of strings on external memory. Another recent paper motivated by database systems is

*Scalable Computation of Acyclic Joins (proceedings of PODS 2006).*

In this paper we show how to perform (acyclic) relational join operations, a cornerstone of modern database systems, in a way that is scalable to a large number of relations. The techniques used during the last 30 years (or so) all suffer from poor performance when the number of relations gets large.