List of Corrections

Ph.D. Thesis
Doctor of Philosophy

DTU Compute
Department of Applied Mathematics and Computer Science

An adventure in data structures

Søren Vind

Kongens Lyngby 2015

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

Write English Abstract

Resume

Write Danish Resume

Preface

This xxx thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a yyy degree in zzz.

The thesis contains a general introduction as well as .....

This dissertation is the result of my research in data structures during my enrollment as a PhD student at DTU Compute.

About my project, collaboration, funding, etc

We vs I

Acknowledgements

- advisors
- external visit
- friends and family
- office
- section
- institute
- university
- coauthors: Philip Bille, Patrick Hagge Cording, Roberto Grossi, Inge Li Gørtz, Markus Jalsenius, Giulia Menconi, Nadia Pisanti, Benjamin Sach, Frederik Rye Skjoldjensen, Roberto Trani, Hjalte Wedel Vildhøj

Contents

CHAPTER 1

Introduction

Data Structures are the basic building blocks in software engineering; they are the organizational method that allow us to store and access information in our computers efficiently. An *Algorithm* specifies the steps we perform to complete some task on an input in order to produce the correct output, relying on underlying data structures. Naturally, deep knowledge and understanding of algorithms and data structures are core competences for software engineers, absolutely vital to developing efficient and predictable software.

In this chapter I will give a brief primer to the study of data structures, an overview of the important problems in the field, and introduce the included papers.

1.1 Bits of Background and Context

In his pioneering work *"On Computable Numbers, with an Application to the Entscheidungsproblem"* [**turing1936computable**] from 1936, Alan Turing in a single stroke became the father of the field of *theoretical computer science*. He gave the first general model of the theoretical capabilities of computers with the *Turing Machine*, and (among others) gave a formalisation of *algorithms*. In the following 80 years theoretical computer science naturally expanded, with the computer revolution and the first computer science institutions being established almost exactly fifty years ago[1]. However, research in the *design and analysis of algorithms and data structures* remain extremely important. The core of the field is:

> Our objective is to use resources efficiently. We *design* data structures and algorithms that solve a *problem*, and *analyse* proposed designs in a *machine model* that allows us to predict and compare the efficiency of different solutions on real computers.

The foundation is the modern equivalents of the concepts introduced by Turing:

Machine Model We use an abstract model of a computer that ignores most details and allows us to understand its behaviour and reason about performance. For example, computation in the very common Word RAM model resembles the capabilities of modern day CPUs: memory is modeled as a sequence of words with w bits. We can read or write a word in unit time and perform arithmetic and word-operations on a constant number of words in unit time.

[1] The Department of Computer Science at Carnegie Mellon University was founded in 1965, as possibly the first such department in the world.

Algorithms In its most general formulation, an algorithm is a method for solving a problem on an input that produces the correct output. The *problem* is a central concept which states the desired properties of the input and output. One classic example of an algorithmic problem is sorting: given as input a list of numbers, return the same numbers in non-decreasing order.

Data Structures A data structure is a method for maintaining a representation of some data while supporting a set of *operations* on the data: allow *updates*, and answer *queries* on the updated data. An example of a data structure problem is sorted list representation: store a list of numbers subject to insertions and deletions, and support queries for the k'th smallest number.

We characterise proposed solutions using several parameters. First, the output of an algorithm or query must always be (one of the possible) *correct* answers. The *performance* of a solution is the amount of resources required to execute it in our machine model, calculated relative to the size n of a finite input: the *space usage* is the number of words (or bits) required, and the *time* is the number of machine operations necessary. We generally consider *worst-case analysis*, meaning the performance obtained when given the input that cause the solution to perform as poorly as possible.

Traditional *deterministic* solutions have inherently bad worst-case performance when solving some problems. To this end, *randomized* solutions relaxes the strict requirements on correctness and worst-case analysis. Instead, *monte carlo* solutions have worst case performance guarantees but some probability of giving an incorrect output. On the contrary, *las vegas* solutions always give a correct output, but with a risk of bad performance.

Generally, research in a particular problem takes two different angles. One is to prove that it is impossible to solve the problem using less resources than some *lower bound* for *any* solution in a given machine model. The opposite direction is to show the existence of a solution where the amount of used resources can be limited by some *upper bound*.

The Classic Example of Sorting It is easy to show a lower bound of $\Omega(n \lg n)$ for sorting n numbers using comparisons and swaps of pairs of numbers: If all n numbers are distinct they have $n!$ possible permutations, only one of which is the correct ordering. Then the number of number-pair comparisons to identify a single permutation uniquely is $\Omega(\lg(n!)) = \Omega(n \lg n)$. There are several matching upper bounds in the form of algorithms solving the problem using $O(n \lg n)$ comparisons[2].

Observe that the sorting lower bound immediately implies a lower bound for any sorted list representation data structure $\mathscr{D}$. The argument is as follows: First insert the entire list of n numbers into $\mathscr{D}$, and then repeatedly get, store, and remove the smallest number in $\mathscr{D}$. Then the result is an ordered version of the original list, created in $O(n)$ operations, meaning that at least one of the operations must take time $\Omega(\lg n)$.

However, the lower bounds does not apply in the Word RAM model where there *is* a linear ($O(n)$) time solution if the numbers each fit in a word. That is, generally lower bounds in weaker models may be circumvented by a stronger machine model.

1.2 The Motivation and Focus for This Work

Rewrite this entire thing

Clearly, although I have only touched upon major concepts there are many possible research directions to take. My main interest is in designing data structures from a theoretical perspective in order to obtain new and improved upper bounds compared to previously known results. I have particularly enjoyed considering the simplest of questions, focusing on core data structure primitives and extremely classic problems.

A common theme in my work and a particular interest of mine is to find solutions that require relatively little space. Each of the proposed data structures exhibit this particular feature, though they span several different classic subfields of algorithmic research, namely combinatorial pattern matching, geometry, and compression.

The papers included in the latter chapters of this thesis each contribute at least one non-trivial newly-designed data structure for solving a particular problem. In some instances, the data structure is used as a cornerstone in giving a solution to a particular algorithmic problem, while in other cases the data structure is the singular contribution. All but one of the papers are purely theoretical, meaning that although the proposed solutions may be good in practice, their efficiency have only been established in theory. In a single case, the proposed data structure was implemented and tested experimentally in practical scenarios.

As initially stated, I consider data structures to be some of the most fundamental building blocks for efficiently solving problems. To see why, consider the following example from the origins of the field of combinatorial pattern matching. In 1970, a fair amount of research had gone into finding solutions to the problem of determining the *longest common substring*[3] of two strings of length n, and thus it was conjectured that no linear time solution existed[4]. However, in 1973 Peter Weiner [**weiner1973linear**] disproved the conjecture by giving a linear time algorithm for building the *suffix tree* data structure, which immediately solved the problem (and many others). In the following 40 years, the properties of the data structure and its extensions have been exploited to solve countless problems[5] in combinatorial pattern matching, where it remains a key primitive.

1.3 Data Structure Primer

In the following, I will give a brief overview of some of the fundamental data structure problems that are relevant for the later chapters, and provide a background for how our results fit in the larger world of data structures. The chapter is meant to give a brief overview of the field from our perspective, and does not provide an exhaustive history. A much more expansive account of our contributions can be found in the next chapter.

[3] Given two strings of length n, what is the longest substring that is in both strings?

[4] donald knuth

[5] many papers

1.3.1 Integer Data Structures

One of the most fundamental and important fields of data structures is that of *integer data structures*, where we operate on a sequence of integers, here denoted X. We will focus on two core problems, called *predecessor* and *prefix sums*.

Predecessor We must store X in order to support the *predecessor* query $\text{PRED}(q)$, asking for the largest element in X that is smaller than q (a *successor* query for the smallest element larger than q must also be supported). Early data structures for the problem only supported that query, but later developments also included support for insertions and deletions in X as well as other closely related queries. The latest solutions are extremely advanced data structures; they are the culmination of decades of work and essentially optimal for all possible parameters[6], yielding a query time of $O(\lg\lg|X|)$ include fusion tree bounds.

Predecessor data structures are used in several of the included papers as a black box. But we also introduced a new data structure in fingerprints for answering *finger predecessor* queries: The query is as before except that it also gives a reference to an existing element $\ell \in X$. For that variant, we give a dynamic solution with query time $O(\lg\lg|\ell - q|)$ number of elements between q, ℓ?, which is much better than the general solution when the reference is close to the query point.

Prefix Sums We must store the sequence of integers X to support three operations: *update* the value of an integer at some position by adding some value to it, find the *prefix sum* of the first i elements, and *search* for the smallest prefix sum larger than some query integer q. It is natural to consider the search operation as a successor query among the prefix sums in X.

The prefix sums problem is extremely well-studied, with a long line of papers showing improved upper and lower bounds[7]. It was shown early that $O(\lg n/\lg\lg n)$ time per operation is sufficient[8]. Due to Patrascu and Demaine[9], we now have matching upper and lower bounds of $O(\lg n/\lg(w/\Delta))$ check this time per query in the Word RAM model, where Δ is the maximal number of bits allowed in the update argument and w is the maximal number of bits per integer in X. In DynamicRelative we give a new improvement to the existing (optimal-time) data structure that also allow insertions and deletions in the sequence X. All previous data structures that allow modifications in X only worked for very small integers (where $w = O(\lg\lg n)$) check this.

1.3.2 Range Searching

In *orthogonal range searching*, we must store a set of points P to support *reporting* and *counting* queries. The input to a query is a rectangle, and the answer is the list of points

[6] vEB, PatrascuThorup, Fusion Trees
[7] ManyPapers
[8] somePaper
[9] PatrascuDemaine, Others?

in P that are contained in the rectangle (or the number of points). If we allow updates, they are typically in the form of point insertions or deletions.

Observe that if the points are in a single dimension we can solve the problem using a predecessor data structure as follows: the query identifies two ends of an interval for which we can find the end points using predecessor queries, and we must report the points between the ends.

In two or more dimensions, there are multiple data structures for range searching.

- How do we answer such queries and keep the point set compressed? - What if the points have colors? - What if the points are moving? - In practice: What about points on the grid that are moving and updated?

Basic Techniques - Trees, DAGs - Fingerprints - Heavy Paths

Advanced Data Structures v- Predecessor (Finger Predecessor) v- Prefix Sums (Dynamic Prefix Sums) - Range Searching (Colored Range Searching) - Pattern Matching - Pattern Extraction

1.4 Background: A Data Structure Primer

- Models (Pointer Machine, Word Ram, Streaming) - Theory vs Practice - Massive Data Sets
- Basics of Data Structures - Trees and Tries - Predecessor
- Strings - Suffix Tree - Fingerprints
- Geometric - Orthogonal Range Searching - With Colors - And Motion
- Compression - DAG compression - SLPs and derivatives

1.5 Our Contributions: An Overview and Outline of Results

The remaining chapters of this dissertation include the following papers, each of them submitted to or published in peer-reviewed conference proceedings. The papers appear in their original form, meaning that notation, language and terminology has not been changed and may not be consistent across chapters. The chapter titles are equal to the original paper titles. Authors are listed alphabetically as is the tradition in the field (except for the paper *Indexing Motion Detection Data for Surveillance Video* as it was published at a multimedia conference where that is not the norm).

Fingerprints in Compressed Strings. By Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Benjamin Sach, Hjalte Wedel Vildhøj and Søren Vind. Presented at Algorithms and Data Structures Symposium (WADS), 2013.

Colored Range Searching In Linear Space. By Roberto Grossi and Søren Vind. Presented at Scandinavian Symposium and Workshops on Algorithm Theory (SWAT), 2014.

Indexing Motion Detection Data for Surveillance Video. By Philip Bille, Inge Li Gørtz and Søren Vind. Presented at IEEE International Symposium on Multimedia (ISM), 2014.

Output-Sensitive Pattern Extraction in Sequences. By Roberto Grossi, Giulia Menconi, Nadia Pisanti, Roberto Trani and Søren Vind. Presented at Foundations of Software Technology and Theoretical Computer Science (FSTTCS), 2014.

Compressed Data Structures for Range Searching. By Philip Bille, Inge Li Gørtz and Søren Vind. Presented at International Conference on Language and Automata Theory and Applications (LATA), 2015.

Annotated Data Streams with Multiple Queries. By Markus Jalsenius, Benjamin Sach and Søren Vind. In submission.

Dynamic Relative Compression By Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj and Søren Vind. In submission.

The journal version of the following paper appeared during my PhD, but as the results were obtained and the conference version published prior to starting the PhD programme the paper is omitted from this dissertation.

String Indexing for Patterns with Wildcards. By Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj and Søren Vind. Presented at Scandinavian Symposium and Workshops on Algorithm Theory (SWAT), 2012. In Theory of Computing Systems, 2014.

For each paper or subject considered in the later chapters, make a subsection. Then in that subsection clarify:

- Our Contributions
- Future Directions

{2.7182818284}ϑφέρτυθιοπσδφγηξκλ