Derandomization, Hashing and Expanders

Milan Ružić

A PhD Dissertation
Presented to the Faculty of the IT University of Copenhagen
in Partial Fulfillment of the Requirements for the PhD Degree

August 2009

Abstract

Regarding complexity of computation, *randomness* is a significant resource beside time and space. Particularly from a theoretical viewpoint, it is a fundamental question whether availability of random numbers gives any additional power. Most of randomized algorithms are analyzed under the assumption that independent and unbiased random bits are accessible. However, truly random bits are scarce in reality. In practice, *pseudorandom generators* are used in place of random numbers; usually, even the seed of the generator does not come from a source of true randomness. While things mostly work well in practice, there are occasional problems with use of weak pseudorandom generators. Further, randomized algorithms are not suited for applications where reliability is a key concern.

Derandomization is the process of minimizing the use of random bits, either to small amounts or removing them altogether. We may identify two lines of work in this direction. There has been a lot of work in designing general tools for simulating randomness and making deterministic versions of randomized algorithms, with some loss in time and space performance. These methods are not tied to particular algorithms, but work on large classes of problems. The central question in this area of computational complexity is "P=BPP?".

Instead of derandomizing whole complexity classes, one may work on derandomizing concrete problems. This approach trades generality for possibility of having much better performance bounds. There are a few common techniques for derandomizing concrete problems, but often one needs to specifically design a new method that is "friendlier" to deterministic computation. This kind of solutions prevails in this thesis.

A central part of the thesis are algorithms for deterministic selection of hash functions that have a "nicely spread" image of a given set. The main application is design of efficient *dictionary* data structures. A dictionary stores a set of keys from some universe, and allows the user to search through the set, looking for any value from the universe. Additional information may be associated with each key and retrieved on successful lookup. In a *static* dictionary the stored set remains fixed after initialization, while in a *dynamic* dictionary it may change over time. Our static dictionaries attain worst-case performance that is very close the expected performance of best randomized dictionaries. In the dynamic case the gap is larger; it is a significant open question to establish if a gap between deterministic and randomized dynamic dictionaries is necessary.

We also have a new analysis of the classical *linear probing* hash tables, showing that it works well with simple and efficiently computable hash functions.

Here we have a randomized structure in which the randomness requirements are cut down to a reasonable level. Traditionally, linear probing was analyzed under the unrealistic *uniform hashing* assumption that the hash function employed behaves like a truly random function. This was later improved to explicit, but cumbersome and inefficient families of functions. Our analysis shows that practically usable hash functions suffice, but the simplest kinds of functions do not.

Apart from dictionaries, we look at the problem of *sparse approximations* of vectors, which has applications in different areas such as *data stream* computation and *compressed sensing*. We present a method that achieves close to optimal performance on virtually all attributes. It is deterministic in the sense that a single measurement matrix works for all inputs.

One of our dictionary results and the result on sparse recovery of vectors share an important tool, although the problems are unrelated. The shared tool is a type of *expander graphs*. We employ bipartite expander graphs, with unbalanced sides. For some algorithms, expander graphs capture all the required "random-like" properties. In such cases they can replace use of randomness, while maintaining about the same performance of algorithms.

The problems that we study require and allow fast solutions. The algorithms involved have linear or near-linear running times. Even sublogarithmic factors in performance bounds are meaningful. With such high demands, one has to look for specific deterministic solutions that are efficient for particular problems; the general derandomization tools would be of no use.

Acknowledgments

When it comes to my PhD studies, I am above all thankful to my advisor Rasmus Pagh. I came to ITU specifically to work with Rasmus on topics of hahsing, and now when my PhD is over, I am glad I did so. Beside making me understand the field better, he provided guidance and support throughout my PhD. In the final years, when I was more independent, he was patient with my long-term projects.

I would like to thank all the people from the *Efficient Computation* group. It was also good being around several people from the *Programming, Logic, and Semantics* group, and it was nice having Ebbe Elsborg and Mikkel Bundgaard as office-mates.

I am grateful to Piotr Indyk for being my host at MIT, where I spent one semester. I learned a lot of new and interesting things, and I enjoyed talking to and being around very enthusiastic people.

Last, but not least, I want to thank my family. My parents, Sreten and Dragica, gave me a great upbringing and support throughout my life.

Milan Ružić
20 August 2009

Contents

x

Chapter 1

Introduction

The first generation of electronic computers was primarily used to perform scientific computations for military use. Not much later, people realized advantages of representing and storing information in digital form. The 1960s saw an increasing use of computers in information management. Storage, retrieval, and communication of data have since then become a very significant, if not dominant, application of computers.

One of basic query types in database systems is to select records that on a particular attribute match a given value. This simple type of query is ubiquitous, sometimes being an entire query, sometimes a part of a more complex query. Not less important is use of this kind of search within numerous other information processing methods. Abstract data structure that directly addresses this basic searching problem is called *dictionary*. The name of the structure reflects a clear association to classical natural-language dictionaries, which comprise a set of words with additional information given for each word, such as meaning, pronunciation etc.

The dictionary problem is very simple to state and there are simple solutions with reasonable performance. Yet, because of its so frequent use in algorithms and data structures, and directly as an indexing structure, it is of interest to have dictionary structures of outstanding performance. Along with continuous advances in computer hardware, growing is the amount of data that needs to be handled. In some computational procedures dictionaries even cause the performance bottleneck.

A big part of this thesis is devoted to improving performance of *deterministic* dictionaries through development of new *hashing techniques*. While notable progress has been made, the problem remains unsettled. Very little is known about the necessary complexity of deterministic dictionary structures, less than for some "richer" problems. Therefore, it is hard to say how close we are to the optimal performance.

In this chapter we review some (basic) background information. In Section 1.1 we say something about families of hash functions. Section 1.2 gives a summary about *expander graphs*, which have use as a derandomization tool. The dictionary problem and the problem of *sparse approximation* of vectors, which is also studied in the thesis, are introduced in Section 1.3. The relevant models of computation are presented in Section 1.4. Finally, in Section 1.5 we summarize

the results contained in the thesis.

1.1 Families of Hash Functions

1.1.1 The family of all functions

Let h be a function chosen uniformly at random from the set of all functions with domain U and range R. We denote $r = |R|$. For any $x \in U$ we may view the value $h(x)$ as a random variable. For any distinct elements $x_1, x_2, \ldots, x_m$ from U, the variables $h(x_1), \ldots, h(x_m)$ are fully independent, that is,

$$\Pr\left\{ \bigwedge_{i=1}^{m} h(x_i) = y_i \right\} = \prod_{i=1}^{m} \Pr\{h(x_i) = y_i\} = 1/r^m \ , \qquad (1.1)$$

for any sequence $y_1, \ldots, y_m$ of elements of R. Let S be any subset of U of size n. For any $y \in R$, the probability that there exists $x \in S$ such that $h(x) = y$ is $\sum_{i=1}^{n} (-1)^{i-1} \binom{n}{i} \frac{1}{r^i}$, by inclusion-exclusion. As a result, the expected size of $h(S)$ is $n - \frac{n(n-1)}{2r} + \frac{n(n-1)(n-2)}{6r^2} - \ldots$ Putting $n = \alpha r$, we see that $\mathrm{E}(|h(S)|)$ is somewhat larger than $n(1 - \frac{\alpha}{2})$. All elements of S that are mapped to the same value $y \in R$ are said to form a chain. The expected length of the longest chain is $\Theta(\frac{\log n}{\log \log n})$ [Gon81].

Classical analyses of hashing methods were done under an idealized *uniform hashing* assumption that the hash function employed behaves like a truly random function. The practical performance of simple hashing heuristics, which do not have theoretical foundation, often follows the predicted performance of the same methods analyzed under the uniform hashing assumption. Yet, use of heuristics occasionally creates problems in practice as well. From a theoretical point of view, it is not satisfactory to rely on such an unrealistic assumption. A lot of work has been done on investigating explicit, and much smaller families of functions. When using feasible families of functions, some hashing algorithms have the same or similar performance as with uniform hashing. Results shown for uniform hashing can be useful for theoretical work, as they provide a reference point and a goal to reach.

1.1.2 Universal families

A family $\mathcal{H}$ is called *c-universal* if for any distinct $x, y \in U$, $\Pr\{h(x) = h(y)\} \leq c/r$, over random choices of $h \in \mathcal{H}$. The notion of universality was introduced by Carter and Wegman [CW79]. Uses of universal hash functions go beyond data structures. Many algorithms make use of hash functions that are expected to have a small number of *collisions* on a set $S \subset U$. If no further knowledge about the distribution of values $h(x)$, $x \in S$, is needed, it is sufficient to employ a universal class of functions. The expected number of collisions will be at most $\binom{n}{2} \frac{c}{r}$. The number of collisions also gives an upper bound on $n - |h(S)|$.

The family of all functions from U to R is, of course, 1-universal. More interesting are explicit families of functions that are efficiently computable. We list a few examples of universal families. With $U = \{0, \ldots, u-1\}$ and $R =$

$\{0, 1, \ldots, r-1\}$, one class is

$$\{x \mapsto (ax \bmod p) \bmod r \mid a \in (\mathbb{F}_p)^*\} \ ,$$

where p is a fixed prime larger than u (see [FKS84], although they they do not explicitly refer to universal classes). Another universal class of functions, with $U = \{0, \ldots, 2^w - 1\}$ and $R = \{0, \ldots, 2^v - 1\}$, is

$$\{x \mapsto (ax \bmod 2^w) \operatorname{div} 2^{w-v} \ : \ a \in U, a \text{ odd}\}$$

It was analyzed in [DHKP97]. These functions do not actually require integer division. A more general form of these multiplicative functions is

$$x \mapsto \lfloor r \cdot \operatorname{frac}(ax) \rfloor \ , \tag{1.2}$$

where $\operatorname{frac}(x)$ denotes $x - \lfloor x \rfloor$. It appears already in [Knu73]. In Chapter 4 we study functions of type (1.2), in a new way.

Now view elements of U as d-dimensional vectors over $\mathbb{F}_p$. For an element x we write $x = (x_1, x_2, \ldots, x_d)$, $x_i \in \mathbb{F}_p$. It is easy to check that the following family is 1-universal.

$$\left\{ x \mapsto \sum_{i=1}^{d} a_i \cdot x_i \mid (a_1, \ldots, a_d) \in (\mathbb{F}_p)^d \right\} \tag{1.3}$$

The multiplications are in $\mathbb{F}_p$ (that is, mod p if the components are viewed as integers). The size of this family is p^d. A related case is when the components of x and a come form a bounded integer domain $\{0, \ldots, 2^v - 1\}$, and we use ordinary integer multiplications in functions $x \mapsto \sum_{i=1}^{d} a_i x_i$. The set of such functions is also universal, although with a slightly weaker constant $c > 1$. The functions of this "dot-product" form appear in Chapter 2. We show how to efficiently pinpoint a function that is as "good" on a given set S as a random function from the class. We do not explicitly mention universality in Chapter 2, as we specifically work with *perfect* hash functions.

1.1.3 k-wise independent families

Just limiting the number of collisions in insufficient for some applications of hash functions. A stronger property than simple universality is independence of hash values. Instead of full independence (1.1), which is typically unnecessary, we may consider families that give limited independence of small order. A family $\mathcal{H}$ of functions is k-wise independent if for any k distinct elements $x_1, \ldots, x_k \in U$ and h chosen uniformly at random from $\mathcal{H}$

$$\Pr\left\{ \bigwedge_{i=1}^{k} h(x_i) = y_i \right\} = \prod_{i=1}^{k} \Pr\{h(x_i) = y_i\} \le c/r^k \ ,$$

for any sequence $y_1, \ldots, y_k$ of elements of R and constant $c \ge 1$. Such families have also been called c *strongly k universal* or (c, k)-universal.[1]

---

[1]Some authors have called (c, k)-universal families that satisfy something similar to k-wise independence, with only an upper bound on probabilities. That is, satisfying only $\Pr\{\bigwedge_{i=1}^{k} h(x_i) = y_i\} \le c/r^k$.

The set of all polynomials of degree $k - 1$ over finite field $\mathbb{F}_p$ is a $(1, k)$-universal family of functions with $U = R = \mathbb{F}_p$. If we identify the elements of $\mathbb{F}_p$ with integers $0, \ldots, p - 1$ and compose each polynomial function with the function $x \mapsto x \bmod r$, we get a family with range $R = \{0, \ldots, r - 1\}$ that is $(1 + r/p)^k$ strongly k universal.

In this thesis, strongly universal classes appear in Chapter 5. We study the performance of hash tables with linear probing under hash functions from families of small independence.

1.2 Expander Graphs

Expanders are graphs that are sparse yet highly connected. The study of expansion properties began in the early 1970s. Over time, it became clear that the significance of expander graphs goes far beyond graph theory. They were found useful in several areas of computer science and mathematics. Unsurprisingly, expansion properties are important in design of robust communication networks. Less obvious are close connections to pseudorandom generators, randomness extractors, and error-correcting codes. A unified view on these objects began to form since late 1990s. Expander graphs also have applications in algorithms and data structures, as we will see in this thesis. In mathematics they play a role, for example, in study of metric embeddings. For a thorough discussion of expanders and list of references see the survey [HLW06]. We will make a brief introduction here.

Graph expansion is often defined in combinatorial terms, but closely related notions can be defined in algebraic or probabilistic terms as well. In the general case we consider undirected and d-regular graphs. In combinatorial terms, we may look at *edge expansion* or *vertex expansion* of a graph. A d-regular graph $G = (V, E)$ is a δ-edge-expander if for every set $S \subset V$ of size at most $\frac{1}{2}|V|$ there are at least $\delta d|S|$ edges connecting S and $\overline{S} = V \setminus S$. To see the algebraic characterization, let A be the adjacency matrix of G, that is, the $n \times n$ matrix, with A_{ij} being the number of edges between vertex i and vertex j. Since A is a real and symmetric matrix, it has n real eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. Let U_1 be the eigenspace of value λ_1. It is easy to see that $(1, 1, \ldots, 1) \in U_1$ and $\lambda_1 = d$. Further, $\dim(U_1) = 1$ (or equivalently $\lambda_1 > \lambda_2$) iff G is connected. It holds that $\lambda_2 = \max_{x \perp (1,\ldots,1)} \langle Ax, x \rangle / \langle x, x \rangle$. The following inequalities, which relate λ_2 to the expansion parameter δ, are well known.

$$\tfrac{1}{2}(1 - \lambda_2/d) \leq \delta \leq \sqrt{2(1 - \lambda_2/d)}$$

A high expansion (or small λ_2) implies another random-like property of G. Namely, for any two sets $S, T \subset V$ the number of edges between S and T is close to $\frac{d}{n}|S||T|$, which equals the expected number of edges between S and T in a random graph with edge density d/n. A probabilistic notion of expansion is based on random walks on graphs. The distribution induced on V by the random walk on an expander G rapidly converges to the uniform distribution.

Expanders are more interesting and useful when d small. Probabilistic arguments easily show that there exist constant-degree expanders. However, it is

not easy to verify if a given graph is indeed an expander. It is of interest to have *explicit* and efficiently constructible expander graphs. There are two (standard) levels of explicitness of constructions of such graphs. The weaker level requires the n-vertex graph to be constructible in time $n^{O(1)}$. The stronger level requires that given any vertex v and $i \in \{1, \ldots, d\}$, the ith neighbour of v can be computed in time poly($\log n$). Such a construction is said to be *fully explicit*. In applications that use the entire graph, it may be enough to have the weaker explicitness. In applications that use only small parts of the entire graph, the stronger level is more adequate.

As an example of a fully explicit family of constant-degree expanders, we state the construction from [Mar73] and [GG81]. The family contains graphs of order m^2, for every integer m. The vertex set is $V_m = \mathbb{Z}_m \times \mathbb{Z}_m$. The neighbours of vertex (x, y) are $(x + y, y)$, $(x - y, y)$, $(x, y + x)$, $(x, y - x)$, $(x + y + 1, y)$, $(x - y + 1, y)$, $(x, y + x + 1)$, $(x, y - x + 1)$ (all operations are in the ring $\mathbb{Z}_m$). These graphs are 8-regular.

For any vertex subset $S \subset V$, define $\Gamma(S) = \{y \in V \mid (\exists x \in S) \{x, y\} \in E\}$. The property of vertex expansion is that $|\Gamma(S) \setminus S| \geq \delta d |S|$, for any $|S| \leq \alpha |V|$ and fixed α, δ. Vertex expansion is a somewhat stronger notion than edge expansion. In particular, if $\alpha = 1/2$ then the graph is clearly a δ-edge-expander as well.

Some applications of expanders require the graph to be bipartite. Further, it is often sufficient that subsets of only one side of the bipartition have large neighbour sets. In a bipartite graph $G = (U, R, E)$, we may respectively call U and R as the "left" part and the "right" part; a vertex belonging to the left (right) part is called a left (right) vertex. We denote the sizes of U and R by u and r. A bipartite graph is called *left-d-regular* if every vertex in the left part has exactly d neighbors in the right part. Such a graph may be specified by a function $\Gamma : U \times [d] \to R$, where $\Gamma(x, i)$ gives the ith neighbour of x. A bipartite, left-d-regular graph $G = (U, R, E)$ is a (N, δ)-expander if any set $S \subset U$ of at most N left vertices has at least $\delta d |S|$ neighbours. Some applications require bipartite expanders with highly unbalanced sides, so that $r \ll u$. Intuitively, it seems harder to achieve good expansion with highly unbalanced sides, because there is less space in which to expand. Ideally we would like to simultaneously have a small degree d, high expansion factor δ, and a value of N that is "close" to r. Using the probabilistic method one can show that there exist $(N, 1 - \varepsilon)$-expanders with $d = O(\frac{1}{\varepsilon} \log \frac{u}{N})$ and $r = O(N \cdot d / \varepsilon)$. The parameter ε cannot be smaller than $1/d$, as some vertices must share neighbours. Logarithmic degree is known to be necessary when $r = O(N \cdot \delta d)$.

No explicit constructions with the aforementioned (optimal) parameters are known. It is a longstanding open problem to approach the optimal parameters with an explicit construction. Notable progress has been made over time. Currently, the best result achieves left degree $d = O((\log u)(\log N)/\varepsilon)^{1+1/\alpha}$ and right set size $O(d^2 N^{1+\alpha})$, for any fixed $\alpha > 0$ [GUV07]. We note that proofs based on analyzing λ_2 are not able show expansion better than $\delta = 1/2$. That is, spectral methods alone cannot yield strong $(N, 1 - \varepsilon)$-expanders.

We may make an interesting comparison of bipartite expanders with families of hash functions. A bipartite expander may be interpreted as a family of d

functions with domain U and range R. For any set $S \subset U$ of size at most N the average size of $\Gamma(S, i)$, over $i \in [d]$, is at least $\delta|S|$. In other words, on a function chosen randomly among $\Gamma(\cdot, 1), \ldots, \Gamma(\cdot, d)$ the image of S will be "large". This is also true if we take a universal family of functions. The minimal possible size of a c-universal family is $r(\lceil \log u / \log r \rceil - 1)/c$ [Meh84]. This means that $\Omega(\log r)$ random bits are needed to pick a random function from a universal family, as opposed to $\log d = O(\log \log u)$ bits needed to pick $\Gamma(\cdot, i)$. If one needs more than a good bound on the image size, for example, a good upper bound on the number of collisions, then functions coming from expanders are not suitable.

Unbalanced bipartite expanders are an essential tool of the methods from Chapters 6 and 7.

1.3 Applications of Hashing and Expanders

The two main problems targeted in this thesis are the dictionary problem and the problem of sparse approximation of vectors. We introduce them in this section. Other applications of hashing techniques and expander graphs are not covered here.

1.3.1 Dictionaries

Dictionaries are among the most fundamental data structures. A dictionary stores a set S which may be any subset of *universe U*, and it answers membership queries of type "Is x in S?", for any $x \in U$. The elements of S may be accompanied by *satellite data* which can be retrieved in case $x \in S$. Some dictionaries also support more general *predecessor queries*, which find $\max\{y \in S \mid y \leq x\}$. The size of the set S is standardly denoted by n.

A dictionary is said to be *dynamic* if it also supports updates of S through insertions and deletions of elements. Otherwise, the dictionary is *static*. Even static dictionaries are sometimes used as stand-alone structures, but more often they appear as components of other algorithms and data structures, including dynamic dictionaries.

The classical solution to the dictionary problem are balanced search trees. They come in many concrete variants, yet without significant differences in performance. Search trees naturally allow predecessor queries. They can easily be augmented to support some other types of queries, such as *rank* and *select*. All queries, as well as update operations, run in $O(\log n)$ time. It is sufficient to supply a comparison predicate to implement the query procedures. The universe U may be abstractly viewed as an arbitrary totally ordered set. The performance of balanced search trees is optimal for the restricted *comparison-based model*, which they operate in.

To surpass the limitations of comparison-based structures one has to know more about the structure of the universe U. Most often the domain of search keys is a set of numbers or a set of character strings. It is then unnecessarily restrictive to consider U just as an ordered set. Especially for the problem of answering membership questions, it is natural to assume that search keys can be viewed as integers or binary strings; keys can always be encoded in that way.

We distinguish between the cases of finite and infinite universe. For example, it is infinite if $U = \{0,1\}^*$. Not all data structures are efficient when keys are long strings. Ideally, we want to have a dictionary where the cost of each operation is proportional to the cost of reading the key.

The dictionary problem has been well studied. Many solutions have been given, having different characteristics regarding space usage, time bounds, model of computation, and universe in question. A challenge is to simultaneously achieve good performance on all the terms. We consider only dictionaries with realistic space usage of $O(n)$ registers of size $\Theta(\log u)$ bits. In the usual case when u is at least polynomially larger than n, this amount of space is necessary (ignoring constant factors) regardless of presence of satellite data. Some data structures focus on conserving space, using an amount close to the information-theoretic minimum, and paying attention to constant factors. A more common practice in analysis of algorithms and data structures is to neglect constant factors in space and time bounds. In this thesis we do not put the highest priority on space usage, and we will usually not keep track of constant factors. The only exception is Chapter 5, where we analyze the performance of linear probing hash tables, as a function of the *load factor* of the table.

Despite of being among the oldest problems in computer science, our understanding of the complexity of the dictionary problem in realistic models of computation has been far from complete. Designing highly efficient dictionaries without resorting to use of randomness appeared to be a particularly challenging task. Randomized dictionaries reached a stage of high development (yet, some open questions still remain). The progress on deterministic dictionaries was much slower. The goal is to bridge the gap between attainable worst-case performance for deterministic dictionaries and the attainable expected performance for randomized dictionaries. As we will see in Chapters 2 and 3, in the static case there is now only a small gap remaining. We do not have a real improvement in the dynamic case. There is some reason to believe that there has to be a gap in the dynamic case, but still there is no definite proof. It is one of major challenges in data structures research to either significantly improve performance of dynamic dictionaries, or to prove general lower bounds that would definitely establish a gap between deterministic and randomized dictionaries.

Information about previous work can be found in the introductory sections of Chapters 2 through 6. Our results are summarized in Section 1.5.

1.3.2 Sparse approximation of vectors

Suppose that data elements are vectors in a high-dimensional space $\mathbb{R}^n$, and that we are able to map $\mathbb{R}^n$ to a low-dimensional space $\mathbb{R}^m$ in a way that roughly preserves some property of interest. Then we can operate on small *sketches* of given vectors, and conclude something about the original vectors from $\mathbb{R}^n$. It is desirable to use *linear* maps for dimension reduction, that is, to use sketches of form Ax, where A is a $m \times n$ matrix and $x \in \mathbb{R}^n$. A reason is that it is easy to update the sketch value Ax under linear updates to x. Likewise, it is easy to obtain the sketch of $x + y$ given the sketches of x and y.

To illustrate dimension reduction maps, it is well known that for any *finite*

set $S \subset \mathbb{R}^n$ there are (plenty of) linear maps to $\mathbb{R}^{O(\log |S|)}$ that approximately preserve ℓ_2 norm on S. In this example, a metric relation is approximately preserved on a set of points. Reductions of this type have been studied for decades. More recently it was discovered that linear maps may be used for obtaining succinct approximate representations of vectors (or signals). Although m is typically much smaller than n, the sketch Ax contains plenty of useful information about the signal x. The vector Ax is also called the *measurement vector*.

The linearity of the sketching method is very convenient for a wide variety of applications. In the area of *data stream computing* [Mut03, Ind07], the vectors x are often very large, and cannot be represented explicitly; for example, x_i could denote the total number of packets with destination i passing through a network router. It is thus preferable to maintain instead the sketch Ax, under incremental updates to x. Specifically, if a new packet arrives, the corresponding coordinate of x is incremented by 1. This can be easily done if the sketching procedure is linear.

Another area of application is *compressed sensing* [CRT06a, Don06, TLW$^+$06, DDT$^+$08]. In this setting x is a physical signal, such as an image. Measurements Ax describe data acquisition process, done using (analog or digital) hardware. Linearity of measurements corresponds to the hardware capabilities. Compressed sensing is an alternative to classical Nyquist/Shannon sampling. Instead of acquiring the entire signal and then compressing it, the idea is to directly sense a compressed version using a smaller number of measurements. It is wasteful to make n samples to acquire every x_i, $1 \le i \le n$, if "main structure" of the signal can be determined using much fewer samples. Many natural or man-made signals can be well-approximated by sparse vectors, which have a limited number of nonzero coefficients.

Formally, we say that a vector *k-sparse* if it contains at most k nonzero entries. It is natural to pose a problem of exactly recovering k-sparse signals using a sketch of size "around" k. This is generalized to the problem of finding good sparse approximations of *arbitrary* vectors. The goal is to find a vector $\hat{x}$ such that the ℓ_p approximation error $\|x - \hat{x}\|_p$ is at most $c > 0$ times the smallest possible ℓ_q approximation error $\|x - x'\|_q$, where x' ranges over all k-sparse vectors (we denote this type of guarantee by "$\ell_p \le c\,\ell_q$"). Note that for any value of q, the error $\|x - \hat{x}\|_q$ is minimized when $\hat{x}$ consists of the k largest (in magnitude) coefficients of x.

The problem has been subject to an extensive research over the last few years, in several different research communities, including applied mathematics, digital signal processing and theoretical computer science. The goal of that research was to obtain encoding and recovery schemes with low probability of error (ideally, deterministic schemes), short sketch lengths, low encoding, update and recovery times, good approximation error bounds and resilient to measurement noise. We use the term "deterministic" for a scheme in which one matrix A works for all signals x, and "randomized" for a scheme that generates a "random" matrix A which, for each signal x, works with probability $1 - 1/n$. However, "deterministic" does not mean "explicit" — we allow the matrix A to be constructed using the probabilistic method.

Information about previous work can be found in Chapter 7. Most of the known algorithms for recovering sparse approximations of signals from their sketches can be roughly classified as either *combinatorial* or *geometric*. With combinatorial algorithms, the measurement matrix is sparse and often binary. Typically, it is obtained from an adjacency matrix of a sparse bipartite random graph. Although not always explicitly stated, usually the actual requirement is that the graph is a high-quality unbalanced expander graph, which a random graph satisfies with high probability. A general prescription of combinatorial recovery algorithms is to iteratively identify and eliminate "large" coefficients of x in some way.

The geometric approach originates from [CRT06b, Don06], and has been extensively investigated since then. Here the matrix A is dense, with at least a constant fraction of nonzero entries. Typically, each row of A is independently selected from a sub-exponential n-dimensional distribution, such as Gaussian. The key property of the matrix A is the *Restricted Isometry Property* [CRT06b], which requires that $(1-\delta)\|x\|_2 \leq \|Ax\|_2 \leq (1+\delta)\|x\|_2$ for any k-sparse vector x. Then, the recovery process can be accomplished by solving the following linear program.

$$\min \|\hat{x}\|_1 \quad \text{subject to} \quad A\hat{x} = Ax$$

A few methods have mixed characteristics from both approaches.

1.4 Models of Computation

Algorithms and data structures are always analyzed with respect to a chosen model of computation. Doing rigorous analysis on a precise model of a real computer would be prohibitively difficult. Although there are many kinds and variations of real computers, differing in performance of execution, their fundamental qualities rarely change. Therefore, theoretical study is done over *abstract* models of computation. A theoretical model should be relatively simple to make analysis easier, as well as statements and comparisons of results. On the other hand, it should be related to real computers, that is, theoretical findings should be a reasonably good indicator of performance on (related types of) actual computers. It is not easy to find a good balance between these demands.

Many different models of computation have been defined. They are not all equally realistic, but concrete context determines their appropriateness. For example, models where main memory storage is represented as a sequentially movable tape are completely inappropriate for analysis of data structures that support fast on-line searches, while they may be acceptable for general study of limits of computation, within the field of computational complexity. Since such archaic models do not make the study of computational complexity easier, beside being inadequate, it is unclear if they need to be used at all.

Below we give an overview of models that appear in this thesis. We use them to different extents; our main model is the *word* RAM. The listed models cover a large part of algorithmic research.

1.4.1 Word RAM

This is a variation of the classical "random access machine" model. Unlike the classic RAM, whose memory cells may contain arbitrarily large integers, content of memory cells in the word RAM model is bounded to integers from $\{0, 1, \ldots, 2^w - 1\}$. As in the classic RAM, there is a (countably) infinite number of memory cells, with addresses $0, 1, \ldots$ The memory is not assumed to be initialized in any particular way on the start of execution. Computation takes place in a central processing unit that has a constant number of local registers. RAM includes standard load/store instructions, control-flow instructions, as well as instructions for performing arithmetic operations on registers. An *algorithm* is a finite sequence of such instructions.

In the word RAM, the content of a memory cell is often called a *word*. Beside being viewed as an integer, a word can be interpreted as a bit string from $\{0, 1\}^w$. The parameter w is hence called the word length. The word RAM has instructions for left and right shifts, as well as bitwise Boolean operations `AND`, `OR`, and `NOT`. These operations would be unnatural on infinite-length registers, present in the classic RAM. Unfortunately, the set of arithmetic operations that should be included in the word RAM has not been standardized. Therefore we have a few variations of the word RAM model. Addition and subtraction are necessarily included. If we stop here, we have a restricted RAM model, sometimes called *practical* RAM.[2] At the next level we include multiplication, and finally we may include all four basic arithmetic operations. The reason for uncertainty about what arithmetic operations should be among the native instructions is mentioned in the following subsection. When saying "the word RAM model" one usually assumes that at least multiplication is a native instruction.

Cost of operations

The execution of every instruction is usually assumed to take constant time. To be more precise, we may charge each instigation with a cost of 1. In this thesis we adopt this *unit cost* model. In practical terms, this is a fairly good approximation; within our instruction set, there are only minor differences in execution times, especially on processors from the last several years. The relative differences may also change over time. For example, in the 1990s compilers would translate the multiplication $5 \cdot x$ as one shift instruction, followed by an addition. A few years later, this combination became slower than one multiplication instruction.

From a theoretical viewpoint, one might argue that it is unfair to say that all instructions take constant time, independent of w. The reason is that multiplication and division belong to $\mathrm{NC}^1 \setminus \mathrm{AC}^0$, that is, they require logarithmic-depth circuits. The other word RAM operations belong to AC^0, that is, they can be implemented with constant-depth, unbounded-fanin Boolean circuits of size $w^{O(1)}$. Although practical experience does not suggest the possibility of having unbounded fan-in gates in reality, *standard* instructions that can be classified as

---

[2]Whether this is a good name is questionable. Still, since it has been used a few times in the literature, we mention it here.

AC^0 are somewhat faster and simpler to implement on actual computers than instructions like multiplication and division. Theoretical scalability of AC^0 instructions was a motivation for studying the AC^0 RAM model. In this model, arbitrary AC^0 instructions are allowed at unit cost. Although AC^0 RAM in principle has an infinite instruction set, any concrete program will use only a finite number of them. This leads to a feasible RAM variant that may be useful for the concrete problem. If some currently nonstandard instructions prove to be really valuable, at least in such a theoretical study, in future they could be promoted to a standard instruction set.

Operations of reading and writing words into memory cells are also assumed to have unit cost, independent of the memory address. This corresponds well to situations where all data fits into the internal memory of a computer. While this is often the case, there are also many applications that involve massive data sets, which far exceed the size of the main memory. Such instances are better modeled by the external memory models.

Trans-dichotomous assumption

In data structure problems it is usually assumed that the universe U matches the set $\{0, 1, \ldots, 2^w - 1\}$ of elements representable by machine words. This also means that $n < 2^w$, where n is the size of the set stored by the data structure. This is sometimes called the trans-dichotomous assumption, as it bridges the dichotomy between the parameters of the problem and the model. If input keys span $O(w)$ bits, instead of only w, a data structure analyzed under the trans-dichotomous assumption would loose only constant factors in performance. Intermediate results of computation may also freely have $O(w)$ bits, as all operations on arguments of $O(w)$ bits can be realized through a constant number of native instructions. If keys may span a larger (superconstant) number of words, different effects may be seen in different data structures; somewhere performance will scale linearly, somewhere not.

The restriction of the universe to nonnegative integers in range $\{0, 1, \ldots, 2^w - 1\}$ is not as strict as it may seem at first sight. Negative numbers were not specified just for simplicity; most of (if not all) word RAM algorithms could easily be adjusted to the model that also includes negative numbers, encoded in the two's-complement system. Some problems, such as sorting, on floating-point numbers represented in the IEEE 754 standard can be solved using algorithms designed for integer data, even without any changes. Not all problems can be reduced to instances with integer (or bit string) universes. But for membership-style problems, which are a main part of this thesis, restriction to universes of type $\{0, 1, \ldots, 2^w - 1\}$ means no limitation whatsoever.

Some people have argued that the word RAM (with the trans-dichotomous assumption) should be called the "standard model", as it is the de-facto standard for modeling internal memory computation.

Uniformity

Some algorithms make use of constants that depend only on the word length w. We may say that such constants are computed at "compile time". A small

number of natural constants, such as w itself, can be assumed to be hard-wired (i.e. available free of charge in the model). For more specialized constants, which are shared by few algorithms, the preprocessing time should be noted. It is usually algorithms which explicitly employ word-level parallelism that require special constants. Many such constants are easily computable in time $O(\log w)$, which is typically negligible. It is not negligible only in (purely theoretical) cases of very large word lengths, for example $w = 2^{n^{10}}$. Then, the preprocessing time may far exceed the running time of the main algorithm. Superpolynomial preprocessing times, such as 4^w, give reason for concern.

An algorithm is called *weakly nonuniform* if it assumes free availability of special constants which depend only on w (that is, the computation of those constants was not included in the stated time bounds). There are many uniform algorithms. For example, take any comparison-based algorithm.

1.4.2   Real RAM

The real RAM is also a variant of the classical RAM model. The domain of values of memory registers is even wider — it is the set of all real numbers. The four standard arithmetic operations and comparisons can be performed on arguments stored in registers. Sometimes the floor function is also included, but this gives a lot of power to the model. If used just for the purpose of hashing, it is quite reasonable to allow conversion to integers. The real RAM is commonly used in computational geometry. Some methods have problems with accuracy of results when working in bounded-precision arithmetic. On the other hand, when an algorithm or a data structure that works in the word RAM model can be translated to the real RAM, it shows independence from boundedness of the universe.

1.4.3   External memory models

Some applications process massive datasets much larger than the size of the main memory. Large datasets need to be stored in data structures on external storage devices such as disks. The I/O communication between internal and external memory may become the main performance bottleneck, far exceeding the other operations in the process. The theoretical I/O-model was introduced to analyze behavior of algorithms on inputs that exceed the size of the main memory. Even long before the formal definition of the model, I/O performance and development of practically efficient data structures was a key issue in database management systems.

The I/O models specifies the size of the main memory M and the size of disk block B as the numbers of data "items" they can accommodate. The context specifies what items are; they may be for example search keys, or records, or even single bits. All computation takes place at the level of internal memory. Data needs to be moved to and from external memory to be processed. An I/O operation is an operation of reading or writing a block from/to disk. The main performance measures are the number of I/Os committed by an algorithm and the amount of space (disk blocks) used. Sometimes the internal memory

computation time is specified as well; the computation time is analyzed as in the word RAM model. The I/O model is simple and allows relatively simple analysis of algorithms, while capturing (most) important part of the memory hierarchy.

The main memory needs to be sufficiently large. It is normal to assume at least that $M \geq 2B$. One can sometimes see *tall-cache* assumptions, such as $M \geq B^2$.

Cache-oblivious model

A variant of the I/O-model is the cache-oblivious model, where the algorithm does not know the values of M and B, that is, the analysis should be valid for any B and M. In this model, I/Os are assumed to be performed automatically by an optimal offline cache replacement policy. Cache-oblivious algorithms are essentially word RAM algorithms, but analyzed in the I/O model, with arbitrary values of M and B. A cache-oblivious algorithm that works well on a machine with the two-level memory hierarchy, also works well on a machine with a deeper memory hierarchy and different relative parameters between neighbouring levels.

Parallel disk model

A way of increasing performance of I/O systems is to combine several disks in parallel. Parallel storage systems have been a reality in practice for a number of years. In theoretical research, the parallel disk model has received some attention. In this model there are D storage devices, each consisting of an array of memory blocks with capacity for B *data items*. The performance of an algorithm is measured in the number of parallel I/Os, where one parallel I/O consists of retrieving (or writing) a block of B data items from (or to) each of the D storage devices. The model can be seen as an extension of the serial I/O model.

1.4.4 Streaming model

A *data stream* is a sequence of data elements, which is much larger than the amount of available memory. Data elements may be, for example, integer numbers, geometric points, etc. The goal is to compute (often approximately) some function of the data by making only one pass over the stream (somewhere, multiple passes may be allowed). A typical situation where data stream computation may be used is a network router. A router cannot afford to store information about every packet that passes thorough. Yet, various statistics of the network traffic are useful to know.

Memory space allowed for streaming algorithms is sublinear, most often polylogarithmic in the size required to represent the object given by a stream explicitly. Processing time per stream element should be relatively fast. Because a very limited amount of information is kept, we have to allow approximation in the result of computation, for almost every function that one wants to compute. *Randomization* is often necessary to achieve good space bounds. Still, there are problems where good deterministic solutions are possible as well.

We mention one common type of stream data. Let x be a vector variable taking values from $\mathbb{Z}^m$ (or $\mathbb{N}^m$, $\mathbb{R}^m$, etc.). The value of x changes over time through updates of its coordinate values. Every update can be specified in form of a double (i, a), $1 \leq i \leq m$ and a is an integer, which has a meaning of increasing the value of x_i by a. A stream is a sequence of such doubles. The value of x before the first update is initialized to zero vector. We may say that x is implicitly determined by the stream. Questions about different statistics of x may be asked after the stream is processed. For example, we may want an estimate of $\|x\|_p$.

1.5 Summary of the Results in the Thesis

In Chapter 2 we present a new type of hash functions and associated algorithms for injectively mapping a given set of n keys to a set of signatures of $O(\log n)$ bits. The methods are computationally efficient in various models of computation, especially for keys of medium to large lengths. More precisely, when given keys have a length of at least $\log^{3+\epsilon} n$ bits, the algorithms for selecting perfect hash functions have a linear running cost on sorted input. Applications of the technique yield a few new solutions to the dictionary problem and even the predecessor problem. Representative results are: a dictionary with a lookup time of $O(\log \log n)$ and construction time of $O(n)$ on sorted input on a word RAM, and a static predecessor structure for variable and unbounded length binary strings that in the cache-oblivious model has a query performance of $O(\frac{|s|}{B} + \log |s|)$ I/Os, for query argument s. All algorithms are deterministic and use linear space.

Chapter 3 complements Chapter 2, yielding static dictionaries with a constant lookup cost and having worst-case cost of the construction that is proportional to only $\log \log n$ times the cost of sorting the input. Chapter 3 has two parts. One part uses the same type of functions as in Chapter 2, but we gave up the requirement of complete injectiveness, and replaced it with considerably weaker and rather specific properties. These weaker functions are meaningful only within our dictionary construction. With this different notion of "good" functions, we achieved a faster construction time in the case of keys of length $\log^{O(1)} n$ bits. The second part describes a very efficient dictionary for universes of size $n^{O(1)}$. Beside its use in composition with methods that perform *universe reduction*, this case has a significance of its own. We devised a different and more efficient construction algorithm for a known type of hash functions.

A new analysis of the well-known family of multiplicative hash functions can be found in Chapter 4, along with deterministic algorithms for selecting "good" functions from this family. Building on these algorithms, we obtain completely *uniform* dynamic dictionaries that achieve a performance of the following type: lookups in time $O(t)$ and updates in amortized time $O(n^{1/t})$, for an appropriate parameter function t. The result is for the word RAM model. However, it also holds in the real RAM model, which illustrates complete independence from the word size. This is not a feature of any other known hashing method.

Hashing with linear probing is the subject of Chapter 5. Here the goal is

not a full derandomization, but proving that linear probing works well with simple kinds of hash functions, which are space and time efficient. We show that selecting a random function from a 5-wise independent family is enough to ensure constant expected time per operation. A relatively small number of random bits is required for such a selection. On the negative side, we show that families that are only pairwise independent may result in expected *logarithmic* cost per operation. We also present two variations to the linear probing algorithm that have a somewhat better performance, at least in the theoretical sense. The results of Chapter 5 are a joint work with Rasmus Pagh and Anna Pagh.

After a classical topic, like linear probing, we move on to algorithms for the *parallel disk model* in Chapter 6. We describe a deterministic load balancing scheme based on expander graphs. This may be of independent interest, yet we primarily use the load balancing scheme as tool for obtaining efficient dictionaries in the parallel disk model. Our main results show that if the number of disks is $O(\log u)$, which is moderately large, a performance similar to the expected performance of randomized dictionaries can be achieved. Thus, we may avoid randomization by extending parallelism. We give several algorithms with different performance trade-offs. The contents of this chapter is joint work with Mette Berger, Esben Rune Hansen, Rasmus Pagh, Mihai Pătraşcu, and Peter Tiedemann.

Chapter 7 is devoted to the problem of sparse recovery of vectors. Our method achieves close to optimal performance on virtually all attributes. In particular, it is the first scheme that guarantees optimal $O(k \log(n/k))$ sketch length, and near-linear $O(n \log(n/k))$ recovery time *simultaneously*. It also features low encoding and update times, and is noise-resilient. The only drawback of our scheme is the $\ell_1 \leq C\ell_1$ error guarantee, which is known to be weaker than the $\ell_2 \leq \frac{C}{k^{1/2}}\ell_1$ guarantee achievable by some of the earlier schemes. The result is a joint work with Piotr Indyk.

Chapter 2

Making Deterministic Signatures Quickly

Abstract

We present a new technique of universe reduction. Primary applications are the dictionary problem and the predecessor problem. We give several new results on static dictionaries in different computational models: the word RAM, the practical RAM, and the cache-oblivious model. All algorithms and data structures are deterministic and use linear space. Representative results are: a dictionary with a lookup time of $O(\log \log n)$ and construction time of $O(n)$ on sorted input on a word RAM, and a static predecessor structure for variable and unbounded length binary strings that in the cache-oblivious model has a query performance of $O(\frac{|s|}{B} + \log |s|)$ I/Os, for query argument s.

2.1   Introduction

Dictionaries are among the most fundamental data structures. A dictionary stores a set S which may be any subset of *universe U*, and it answers membership queries of type "Is x in S?", for $x \in U$. The elements of S may be accompanied by *satellite data* which can be retrieved when $x \in S$. Some dictionaries also support more general *predecessor queries*, which find $\max\{y \in S \mid y \leq x\}$. The size of set S is standardly denoted by n.

We consider universes whose elements can be viewed as integers or binary strings. In this chapter we concentrate on *static* dictionaries — a static dictionary is constructed over a given set S that remains fixed. *Dynamic* dictionaries allow further updates of S through insertions and deletions of elements. Even static dictionaries are sometimes used as stand-alone structures, but more often they appear as components of other algorithms and data structures, including dynamic dictionaries.

The most important performance parameters of a dictionary are: the amount of required storage space, the time needed for performing a query, and the time spent on an update or construction of the whole dictionary. The dictionary problem has been well studied and many solutions have been given. There

are still a few open questions and places to improve, especially in the area of *deterministic* dictionaries.

A concept which is often encountered in the solutions to the dictionary problem and more general searching problems is *universe reduction*. The goal is to find a mapping from U to a smaller set, with the function being injective on S. Then either continue recursively or make use of a special procedure that takes advantage of the smaller universe.

We present a new method of universe reduction down to a range of size polynomial in n. Every element of S gets a unique *signature* of $O(\log n)$ bits. Conceptually, the reduction process is gradual. Yet, with appropriate combinations of the basic method, it is possible to calculate the result of the final mapping quickly. The function that is the building block of the entire method is rather simple. We show that it is possible to relatively efficiently find parameters of that function that make it injective on a given set of keys.

Applications of the technique yield a few new solutions to the dictionary problem and even the predecessor problem. Our predecessor structures are suitable for variable-length keys. New results can be expressed in different computational models: the *word RAM*, the *practical RAM*, and (for strings) the *cache-oblivious* model. All algorithms are deterministic and use linear space.

Having a dictionary based on signatures can be an advantage in some settings. A weaker form of dictionaries answers not membership but *retrieval* queries: given $x \in S$ retrieve the data associated with x, and arbitrary output if $x \notin S$. We can rephrase this as: retrieve the data associated with the signature of x, if any. An example of a setting where this kind of dictionary can occasionally be useful is a distributed system. If keys are relatively long then we may send a short signature of a key x to a remote data server. If $x \in S$ or the signature of x does not belong to the signature set of S, then the correct answer and data will be returned. Otherwise, data associated to some (unknown) element of S will be returned. When this kind of behavior is acceptable, the use of signatures increases the throughput of the system.

2.1.1 Related work for word RAM

The word RAM is a common computational model in the data structures literature. The parameter w represents the machine word size. A usual assumption for RAM dictionaries is that the elements of U fit in one machine word. Algorithms from this chapter do not use division operation.

Among the dictionaries with constant lookup time, the structure described in Chapter 3 has the fastest construction time, which is proportional to $\log \log n$ times the cost of sorting the input. With the currently fastest sorting algorithm [Han04] this makes a running time of $O(n(\log \log n)^2)$. A part of that solution is actually a follow-up on the technique of universe reduction that is described in the present chapter. Another part is a very efficient dictionary for universes of size $n^{O(1)}$.

At the moment, the fast static dictionaries from Chapter 3 do not yield an improvement for dynamic deterministic dictionaries. Known (generic) dynamization techniques give the same performance as they give with a construction time

of $O(n \log n)$ for static dictionary, which was earlier achieved by Hagerup, Miltersen, and Pagh [HMP01]. A similar performance in the dynamic case has the structure from Chapter 4.

Fusion trees of Fredman and Willard [FW93], which also support predecessor queries, have a good performance when the word size is very large. A generalization of the fusion trees gives a linear-space static dictionary with a query time of $O(1 + \frac{\log n}{\log w})$ and $O(n)$ construction time on sorted input (this was explicitly stated by Hagerup [Hag98b]).

Andersson's *exponential search trees* [And96] are a general method for turning a static polynomial-space predecessor structure into a linear-space dynamic predecessor structure. Beame and Fich [BF02] proved a tight bound of $\Theta(\frac{\log w}{\log \log w})$ for the predecessor problem in polynomial space (when expressed only in terms of w). Plugging fusion trees and the structure of Beame and Fich into the framework of exponential search trees results in a predecessor dictionary with a query time of $O(\min\{\frac{\log w}{\log \log w} \log \log n, \sqrt{\frac{\log n}{\log \log n}}\})$. That structure can be constructed from a sorted list of keys in $O(n)$ time. A result by Pătrașcu and Thorup [PT06] implies that in linear space predecessor queries cannot be answered in time faster than $\Omega(\log w)$.

2.1.2 Related work for practical RAM

The unit-cost assumption for all standard instructions may be regarded as too strong because multiplication requires a Boolean circuit of depth $\Theta(\frac{\log w}{\log \log w})$, when circuit size is $w^{O(1)}$. The presence of an instruction of that circuit complexity was shown to be necessary to achieve constant time lookups [AMRT96]. Some work was done on investigating how efficient dictionaries (and more general predecessor structures) are possible when only "cheap" instructions are used.

Exclusion of multiplication and division from the word RAM leads to a model sometimes called practical RAM. Brodnik, Miltersen, and Munro [BMM97] showed how to achieve a lookup time of $O(\sqrt{\log n}(\log \log n)^{1+o(1)})$ on a practical RAM. They also described some computational routines relying only on the basic instruction set. Of interest to us is multiplication of two words in time $(\log w)^{1+o(1)}$. The routine applies well to field-wise multiplication: $\Omega(w/v)$ products of v-bit integers packed in words can be computed in time $(\log v)^{1+o(1)}$. Special constants must be available, but they can be computed in time $v^{O(1)} \log w$. It is not claimed that doing multiplication this way is practical.

Andersson [And95] showed that in the practical RAM model there is a data structure which enables an arbitrary set of $\Theta(\log n)$ search requests to be answered in $O(\log n \log \log n)$ time; here search means the stronger neighbour search. However, the preprocessing time and the required space are superpolynomial.

2.1.3 Strings and the cache-oblivious model

Real computers don't have one plain level of memory but a memory hierarchy. Transfers of data between levels of memory may be a dominant term in execution times. The theoretical I/O-model was introduced to analyze behavior of

algorithms in such a setting. A variant of the I/O-model is the cache-oblivious model [FLPR99], where the algorithm does not know the size of the internal memory M and the block size B; that is, the analysis of an algorithm should be valid for any values of B and M.

Comparison-based sorting of n integers (which occupy one memory cell) takes $\Theta(\text{Sort}(n))$ I/Os, where $\text{Sort}(n) = \frac{n}{B} \log_{M/B} \frac{n}{B}$ [FLPR99]. The complexity of sorting strings has not been settled even in the I/O-model. Some results on deterministic sorting appear in [AFGV97]. In the cache-oblivious model a simple upper bound is $O(\text{Sort}(N))$, where N is the total number of characters in the strings, and it can be achieved by building a suffix array [KS03] over the concatenation of the strings. A faster external-memory sorting algorithm appears in [FPP06], but it is randomized.

In the I/O-model the *string B-trees* of Ferragina and Grossi [FG99] support lookups and updates in $O(\log_B n + |s|/B)$ I/Os, with s being the argument. This bound is optimal for comparison-based structures and unbounded alphabets. String B-trees crucially depend on knowing the value of B. Using a different technique, Brodal and Fagerberg [BF06] gave a static data structure for the cache-oblivious model with the same search performance of $O(\log_B n + |s|/B)$ I/Os. Their structure can be constructed using $O(\text{Sort}(N))$ I/Os, with N being the total length of the strings. The structure assumes $M \geq B^{2+\epsilon}$, and it is not simple. Bender et al. [BFCK06] introduced a randomized structure called *cache-oblivious string B-trees*, which are dynamic and support also predecessor searches; here the tall-cache assumption is of type $M \geq B^2$.

The string B-trees support *range queries*. A special case of range query is *prefix query*: finding strings that have the query argument as a prefix. The cache-oblivious string dictionary from [BF06] supports prefix searches. However, those data structures exhibit no speed-up when only membership queries are wanted. Using some types of hash functions it is possible to do dictionary lookups using optimal $O(|s|/B)$ I/Os, but no previously known algorithm can deterministically construct an appropriate function using a reasonably low number of I/Os.

2.2 Our Results

Our main technical contribution lies in novel universe reduction techniques. The results we state on dictionaries basically come as corollaries. Other advanced data structures are used as ingredients in the dictionary results, but the universe reduction results are pretty independent. The results on predecessor structures for long strings were made possible primarily due to the new efficient dictionaries for strings; still, they do not come as completely obvious consequences.

Our main contributions can be viewed as a single technique which then gets adjusted for different models. In this section statements of results will be organized according to computational model.

To create dictionary structures based on our reduction method, we need to utilize an efficient dictionary for universes of size $n^{O(1)}$. Currently, the fastest dictionary for that case is given in Chapter 3. It is convenient that we can use it for all three computational models. Since multiplications are not utilized,

there are no differences between word RAM and practical RAM versions of the structure. In the version for the cache-oblivious model only the sorting procedure needs to be changed. The construction cost scales nicely when several keys can be packed in single machine word on a RAM, again by using an appropriate sorting procedure.

We use notation $[x]$ to represent the set $\{0, 1, \ldots, x - 1\}$.

2.2.1 Results for word RAM

We start with a universe reduction result. Here we consider the case where $w \leq n^{O(1)}$. The (unrealistic) situation with $w \geq n^{\Omega(1)}$ can be covered efficiently with fusion trees.

Theorem 2.2.1. *In the word RAM model with a word length of $w \leq n^{O(1)}$ bits, for a given set S of n keys there is a function mapping $[2^w]$ to $[n^{O(1)}]$ that is one-to-one on S, with $O(1)$ evaluation time and description of size $O(1)$ words that can be computed deterministically in time $O(n + n\frac{(\log n)^3}{w}(\log \frac{w}{\log n})^3)$ assuming that the input set of keys is sorted.*

It is apparent that for $w > (\log n)^{3+\epsilon}$ the construction of the function takes linear time on sorted input. The assumption that $w \leq n^{O(1)}$ helps the evaluation procedure, and we are able to state constant evaluation time. In the general case it is possible to achieve an evaluation time of $O(\log \log w)$ with our signature functions. Further, excluding large word sizes allows us to neglect additive terms in construction times of type $\log w(\log n)^{O(1)}$, which would become dominant when w approaches 2^n. The method does not require any special constants that depend on w to be prepared "in advance". All the necessary computations are included in the stated time bound.

By composing the method from Theorem 2.2.1 with the dictionary for small universes from Chapter 3, and combining the resulting structure with fusion trees [FW93, Hag98b] to cover the case $w \geq n^{\Omega(1)}$, we get the result formulated next. Although the construction time of the dictionary for small universes is in general $O(n \log \log n)$, the time is $O(n)$ when $\Omega(\log n(\log \log n)^2)$ input keys can be packed in a word.

Theorem 2.2.2. *For $w > (\log n)^{3+\epsilon}$ there is a deterministic linear-space solution to the static dictionary problem with $O(1)$ lookup time and $O(n)$ construction time on sorted input.*

In Section 2.6 we will see that the reduction method can be adapted to work with binary strings that can span multiple words and can have different lengths. We are also interested in linear-space predecessor structure that accepts unbounded and variable length keys. In Section 2.7, we show how to modify the method of recursion on universe size from the van Emde Boas (vEB) structure [vEBKZ77] to work in that setting.

Theorem 2.2.3. *Let S be a set of n binary strings, and let N be the sum of their lengths in bits. There is a deterministic static predecessor structure that can store S using $O(N)$ bits of space and answer queries in $O(\frac{|s|}{w} + \log |s|)$ time, with*

s being the query argument. On sorted input set, the structure can be constructed in time $O(n + N/w)$.

The best alternatives are relatively simple comparison-based solutions that have a query time of $O(\frac{|s|}{w} + \log n)$. We have an improved query time for arguments of length $O(n)$ bits. In the special case when all keys have a fixed length of w bits, the implementation is considerably simpler. Query time is then $O(\log w)$.[1] An interesting dictionary structure with a very fast construction time can be created by using the predecessor structure for fixed-length keys when $w \leq (\log n)^{3+\epsilon}$, and otherwise using the dictionary from Theorem 2.2.2.

Theorem 2.2.4. *There is a deterministic linear-space solution to the static dictionary problem with $O(\log \log n)$ lookup time and $O(n)$ construction time on sorted input.*

2.2.2 Results for practical RAM

Our primitive type of functions involves one multiplication, and one operand is always a "small" integer. This enables us to get a theoretically interesting result in the practical RAM model, with a particularly good performance when small sets of search requests are given at once. In Section 2.4 we will see that it is relatively easy to derive the algorithms for this model form the corresponding algorithms for the word RAM model. Here we place no restriction on w.

Proposition 2.2.5. *In the practical RAM model with word length w, for a given set S of n keys there is a function $h : [2^w] \rightarrow [n^{O(1)}]$ that is one-to-one on S, with the following properties:*

- *the description of h can be computed in time*

$$O(n \log n (\log \log n)^2 (\log w)^{1+o(1)} + (\log w)^{O(1)})$$

  with a deterministic algorithm;

- *the evaluation time of h on single argument x is $O((\log \log n)^{1+o(1)} \log \frac{w}{\log n})$;*

- *For any set $T \subset U$ of $\Omega(\log \frac{w}{\log n})$ arguments, the values $h(x)$, $x \in T$, can be computed in total time $O(|T|(\log \log n)^{1+o(1)})$.*

A consequence of Proposition 2.2.5 is the following:

Theorem 2.2.6. *In the practical RAM model with word length w, there exists a deterministic linear-space solution to the static dictionary problem with a construction time of $O(n \log n (\log \log n)^2 (\log w)^{1+o(1)} + (\log w)^{O(1)})$, and a lookup time of $O((\log \log n)^{1+o(1)} \log \frac{w}{\log n})$ for single keys. Additionally, for any set $T \subset U$ of size $\Omega(\log \frac{w}{\log n})$, lookups of all elements of T can be committed in total time $O(|T|(\log \log n)^{1+o(1)})$.*

---

[1]The result for fixed-length word-size keys is not really a new thing. It could have been stated as soon as the dictionary from [HMP01] appeared.

An interesting property of this dictionary is efficient processing when query arguments are given in a small batch. In batched evaluation, the cost of processing per element is exponentially lower than the lower bound for single lookups in the stronger AC^0 RAM model [AMRT96]. Previous solutions with asymptotically efficient lookups in the restricted RAM models [BMM97, AMRT96, Hag98a] make use of universal hash functions. Without randomization, their construction times are considerably higher than that of our dictionary.

2.2.3 Results for cache-oblivious model

We again start with a result about signature functions. Since we assume that strings are over binary alphabet, the values of I/O parameters B and M represent quantities in bits.

Theorem 2.2.7. *Let S be a set of n binary strings, and let N be the sum of their lengths in bits. In the cache-oblivious model, there is a function mapping $\{0,1\}^*$ to $[n^{O(1)}]$ that is one-to-one on S, and on sorted input its description can be computed deterministically in*

$$O\left(\frac{N}{B} + \frac{\log^2 n \cdot \log_{M/B}(n/B)}{B} \sum_{s \in S} \log |s|\right) \tag{2.1}$$

I/Os. Evaluation of the function on argument s requires $O(\frac{|s|}{B})$ I/Os. The bound on performance of the construction holds under a tall-cache assumption of type $M > B^{1+\delta}$. The evaluation procedure needs no tall-cache assumption.

The tall-cache assumption is solely due to sorting algorithm. When the average length of the strings from S is $\Omega(\log^{2+\epsilon} n \log_{M/B}(n/B))$ bits, then the value of (2.1) is $O(N/B)$.

The signature mapping can be composed with the dictionary for universes of polynomial size (Chapter 3).

Theorem 2.2.8. *Let S be a set of n binary strings, and let N be the sum of their lengths in bits. In the cache-oblivious model, a dictionary structure can store S using $O(N)$ bits, perform lookups in $O(\frac{|s|}{B})$ I/Os on argument s, and have I/O cost of the construction as in (2.1) assuming that input set is sorted.*

In a similar way that the predecessor structure for strings on a RAM is constructed, we create a predecessor structure for external memory. This is discussed in Section 2.7.

Theorem 2.2.9. *Let S be a set of n binary strings, and let N be the sum of their lengths in bits. There is deterministic static predecessor structure that can store S using $O(N)$ bits of space and answer queries in $O(\frac{|s|}{B} + \log |s|)$ I/Os, with s being the query argument. On sorted input set, the structure can be constructed using $O(N/B)$ I/Os.*

We have an improved query time for arguments of length $O(n)$ bits. The shorter the query string is, the greater advantage we have over the classical $O(\frac{|s|}{B} + \log n)$ bound. For string dictionaries we can also have the following trade-off: lookups in $O(\frac{|s|}{B} + \log \log n)$ I/Os, and construction cost of $O(\frac{N}{B})$ on sorted input.

2.3 Basis of Universe Reduction Method

This section describes our primitive method for reducing a universe of size u to a universe of size "around" $\sqrt{u}$, assuming that $u \geq n^d$ for a sufficiently large constant d. We start in Section 2.3.1 with defining the type of functions that carry out the reduction, and expressing some observations about those functions. Subsequently, Section 2.3.2 outlines the algorithm for selecting an injective function on a given set S. The crucial subprocedure of the algorithm comes in two versions, which are discussed in Sections 2.3.3 and 2.3.4. All other constructions from the chapter use as the building block the method presented in this section.

Model-dependent aspects are presented with a focus on the word RAM model, with occasional remarks about practical RAMs. Adjustments for the external memory models are straightforward. The universe $U = [u]$ is such that $u \leq 2^w$. We use the notation $S = \{x_1, x_2, \ldots, x_n\}$. Also, $\log x$ means $\log_2 x$.

2.3.1 Type of functions

The essential type of functions is

$$f(x, s, a) = x \text{ div } 2^s + a \cdot (x \bmod 2^s) \ ,$$

where a is a parameter chosen from $\{1, 2, \ldots, n^c - 1\}$, $c \geq 2$. The parameter s will have a value dependent only on the domain of x. If x can take any value from U then a suitable choice for s is $\lceil \frac{1}{2} \log u \rceil$. The integer division and modulo functions were chosen as they are perhaps the simplest of all pairs of functions (ϕ, ψ) such that (ϕ, ψ) is 1-1 on U, and so that both functions map to a (significantly) smaller universe. In a more general form, we write $f(x, a) = \phi(x) + a \cdot \psi(x)$. The parameter s is left out when its value is not relevant or is understood.

The selection of a value for the parameter a that makes f injective on S is done through a type of binary search, as we will see in the next subsection. Throughout this section μ will denote the middle point of an interval in the binary search. The algorithms compute the set $\{f(x, \mu) \ : \ x \in S\}$ at every step of the binary search, and make a decision where to continue the search based on some properties of this set.

We see that for $i \neq j$:

$$f(x_i, a) = f(x_j, a) \iff a(\psi(x_i) - \psi(x_j)) = \phi(x_j) - \phi(x_i) \ . \qquad (2.2)$$

If $\psi(x_i) = \psi(x_j)$ then $\phi(x_i) \neq \phi(x_j)$, so no parameter causes a collision between x_i and x_j. Otherwise, there is at most one multiplier for which x_i and x_j collide. The value $\frac{\phi(x_j) - \phi(x_i)}{\psi(x_i) - \psi(x_j)}$ may not be an integer, but possible non-integralities will be disregarded in the binary search; that is, all values $\frac{\phi(x_j) - \phi(x_i)}{\psi(x_i) - \psi(x_j)}$, for different i and j, which are contained in a selected interval are counted, and they will be referred to as the *bad parameters*.

The equivalence stated in (2.2) involves two equalities. Obviously, we can substitute an inequality operator for the equality operator, namely:

$$f(x_i, a) \gtrless f(x_j, a) \iff a(\psi(x_i) - \psi(x_j)) \gtrless \phi(x_j) - \phi(x_i) \ .$$

We apply a straightforward case analysis that shows a relation of the bad parameter for a pair (x_i, x_j), $i \neq j$, to the given value of μ. W.l.o.g. we assume that $\phi(x_i) \leq \phi(x_j)$. Recall that the set of candidates for the multiplier consists only of positive values.

1. If $(\phi(x_i) < \phi(x_j) \wedge \psi(x_i) < \psi(x_j)) \vee \phi(x_i) = \phi(x_j) \vee \psi(x_i) = \psi(x_j)$ then f maps x_i and x_j to different values for any a.

2. If $\phi(x_i) < \phi(x_j) \wedge \psi(x_i) > \psi(x_j)$ then consider the following subcases:

    (a) $f(x_i, \mu) > f(x_j, \mu)$: a collision between x_i and x_j can occur only for a parameter smaller than μ (and larger than 0).

    (b) $f(x_i, \mu) < f(x_j, \mu)$: a collision between x_i and x_j can occur only for a parameter larger than μ.

    (c) $f(x_i, \mu) = f(x_j, \mu)$: f maps x_i and x_j to different values for any $a \neq \mu$.

Define the following two values:

$$
\begin{aligned}
m_1 &= |\{ \{i, j\} \ : \ \phi(x_i) < \phi(x_j) \wedge f(x_i, \mu) > f(x_j, \mu)\}| \ , \\
m_2 &= |\{ \{i, j\} \ : \ \psi(x_i) > \psi(x_j) \wedge f(x_i, \mu) < f(x_j, \mu)\}| \ .
\end{aligned}
$$

According to the case analysis, m_1 counts exactly the pairs which fall under the case 2(a). Thus, it is the number of bad parameters in $(0, \mu)$, which is an upper bound on the number of multipliers from $\{1, \ldots, \mu - 1\}$ for which f is not injective. Analogously, m_2 counts exactly the pairs which fall under the case 2(b). Thus, it is the number of bad parameters in $(\mu, +\infty)$, which is an upper bound on the number of multipliers from $\{\mu + 1, \ldots, n^c - 1\}$ for which f is not injective.

2.3.2 Choosing a good multiplier

There are at most $\binom{n}{2}$ parameters of f which are to be avoided. The binary search for an appropriate value of a starts by setting the left end of the interval to 1, the right end to $n^c - 1$, and μ to be the midpoint. If $f(x, \mu)$ is found to be 1-1 then μ is selected as the parameter and the procedure is finished. The interesting case is when f is not 1-1 and we want to reduce the search space in such a way that the repeated procedure is guaranteed to find a good multiplier.

Since m_1 and m_2 represent the sizes of disjoint classes of bad parameters, we have that $m_1 + m_2 \leq \binom{n}{2}$. We may evaluate both m_1 and m_2, and then choose the half with the smaller corresponding value. Then the search space is reduced to an interval which contains at most $n^2/4$ bad parameters. The same upper bound may be achieved by computing only m_1, and then choosing the lower half if $m_1 < n^2/4$.

Suppose that we are left with an interval $(\bar{l}, \bar{r})$ such that $\bar{l} > 0$. To determine the number of inappropriate parameters inside the lower half, we may subtract from (the new value of) m_1 the number of bad parameters not larger than $\bar{l}$ — that value is accumulated in previous steps of the binary search. Yet, if only approximations of the m_1 values are computed (constant factor approximations)

and not exact values, then this "subtraction method" does not work. We will describe a simple reduction that works in both cases. Since it was decided to select a multiplier from $(\bar{l}, \bar{r})$, the function may be written as $\phi(x) + (\bar{l} + a)\psi(x)$, where a is now to be chosen from $\{1, \ldots, \bar{r} - \bar{l} - 1\}$. The function $x \mapsto (\phi(x) + \bar{l} \cdot \psi(x), \ \psi(x))$ is 1-1 on U as well. Thus, by replacing the $\phi(x_i)$ values with $\phi(x_i) + \bar{l} \cdot \psi(x_i)$, the problem becomes equivalent to the one in the first step of the binary search. Note that in the same way we could reduce the problem of counting bad parameters in the higher half to the problem of counting bad parameters in the lower half of a shifted interval.

After completing $O(\log n)$ steps of the binary search we have isolated a multiplier for which f is injective on S. We will present two algorithms for computing values of type m_1. The first one performs exact computation, while the second one returns approximate values. The selection procedure based on approximate calculations is faster, at a cost of requiring a slightly larger domain of parameters. With the first algorithm we may set c to be as low as 2, whereas the second one allows c to be as low as 3.42. The approximation-based procedure cannot use values of type m_2 because such values have to be combined using subtraction to get the number of bad parameters in the higher half of current interval (recall that m_2 gives the number of bad parameters in an unbounded interval). As there is no real benefit from estimating the number of bad parameters in both halves of an interval, both procedures are set to choose the lower half when (an estimate or exact value of) m_1 is low enough.

2.3.3 Exact computation of m_1

We utilize a dynamic data structure supporting rank queries. For a multiset $\hat{S} \subset \mathbb{R}$ (i.e., $\hat{S}$ may contain duplicates), the rank of a number x with respect to $\hat{S}$ is defined as $rank_{\hat{S}}(x) = |\{x' \in \hat{S} \ : \ x' < x\}|$. The problem of answering rank queries and supporting insertions and deletions on $\hat{S}$ is is known as the *subset rank* problem. An augmented balanced search tree supports all operations in $O(\log n)$ time — every internal node additionally maintains information about the sizes of subtrees rooted at that node. Dietz [Die89] made an improvement to $O(\log n / \log \log n)$ time per operation, which in general matches the lower bound from [FS89]. In our case, deletions do not need to be supported by the subset rank structure, which makes implementation easier.

The method for computing m_1 starts by sorting the set S according to the ϕ values of the elements. Without loss of generality, suppose that the sorted sequence is $(x_1, x_2, \ldots, x_n)$, meaning that $\phi(x_i) \leq \phi(x_j)$ for any $i < j$. As the next step, the multiset $F = \{f(x, \mu) \ : \ x \in S\}$ is sorted; this provides for each value $f(x_i, \mu)$ its rank within F. Denote the resulting values by $rankf(x_i, \mu)$. That way, F is mapped to the set $[n]$ by a monotonically increasing function. The subset rank data structure is initialized so that $\hat{S} = \{rankf(x_n, \mu)\}$. Then, in reverse order, from $n - 1$ to 1, the values $rank_{\hat{S}}(rankf(x_i, \mu))$ are summed and the values $rankf(x_i, \mu)$ are inserted into $\hat{S}$. But if some value $\phi(x_i)$ is not unique among the ϕ-values, the element $rankf(x_i, \mu)$ is not immediately added to $\hat{S}$. Suppose that $(\phi(x_j), \phi(x_{j+1}), \ldots, \phi(x_k))$ is a maximal run of equal ϕ-values, that is, it cannot be extended on left or right. First all the values

$rank_{\hat{S}}(rankf(x_i, \mu))$ are computed, $j \leq i \leq k$, and afterwards the multiset $\{rankf(x_j, \mu), \ldots, rankf(x_k, \mu)\}$ is added to $\hat{S}$. The final output value is equal to the sum of calculated $rank_{\hat{S}}(rankf(x_i, \mu))$ values. This procedure correctly computes m_1.

The performance of the procedure depends on the choices for sorting algorithm and subset rank data structure. One option is to pick the fastest general (deterministic) methods. Naturally, we would choose Han's sorting algorithm [Han04], and Dietz's data structure. In that case, the computation of m_1 would take $O(n \log \log n + n \log n / \log \log n)$ time. However, when several keys may fit into one word, an even better performance can be achieved by using simpler algorithms and data structures that exploit word-level parallelism to a greater extent. Also observe that the subset rank structure works with keys of only $\log n$ bits.

For rank computations, a structure that takes advantage of relatively larger word sizes is the packed B-tree [FW93, And95], augmented to support rank queries. The rank procedure utilizes prefix summation on word level; that cannot be done in constant time on a practical RAM, so these improvements apply to the word RAM model only. In one step of the binary search, all rank queries and insertions into trees take $O\left(n(1 + \log n / \log \frac{w}{\log n})\right)$ time.

As for sorting, we may choose a serial version of the parallel sorting algorithm due to Albers and Hagerup [AH92, AHNR98]. They show that if K keys can fit in a certain word-size structure, then merging two sorted sequences stored in such structures takes $O(\log K)$ time. Merging longer sequences is done using this subroutine instead of going down to single comparisons. In this way a factor of $\Theta(K / \log K)$ is saved comparing to standard merge sort. In our case, K can in principle be as high as n. Even if the word size is very large, we have no use of being able to accommodate more than n keys. Therefore we can put $K = \min\{\frac{w}{\lceil \log u \rceil}, n\}$. If $K = \Omega(n)$ then sorting takes $O((\log n)^2)$ time. Otherwise, when the word size is not so large, the cost of sorting tasks in the computation of m_1 is $O(n \log n \frac{\log u}{w} \log \frac{w}{\log u})$ when the mentioned sorting procedure is used. There is no reason to employ word-level parallelism on other jobs, like computing function values, because rank computations are more expensive anyway.

2.3.4 Substituting rank queries with permutation inversions

Viewing the definition of m_1, one may observe that it resembles counting certain inversions. A problem with reduction to inversions of a permutation is that there may be duplicates among the ϕ-values and the function values. A solution is to define strict linear orders which will separate equal values but will not cause any additional inversions. We define orders $\prec_f$ and $\prec_\phi$ with:

$$x \prec_f y \iff f(x, \mu) < f(y, \mu) \vee (f(x, \mu) = f(y, \mu) \wedge \phi(x) < \phi(y)) \ ,$$
$$x \prec_\phi y \iff \phi(x) < \phi(y) \vee (\phi(x) = \phi(y) \wedge x \prec_f y) \ .$$

Rank functions based on these order relations are $rank_f : S \to [n]$, $rank_f(x) = |\{x' \in S \ : \ x' \prec_f x\}|$, and $rank_\phi : S \to [n]$, $rank_\phi(x) = |\{x' \in S \ : \ x' \prec_\phi x\}|$. The values of the rank functions for the whole set S are calculated by first sorting

the ordered pairs $(f(x_i, \mu), \phi(x_i))$, and then using the produced $rank_f$ values sorting the set of ordered pairs $(\phi(x_i), rank_f(x_i))$. Let π_1 be a permutation of $[n]$ given by $\pi_1(i) = rank_f(rank_\phi^{-1}(i))$. It is easy to check that $m_1 = Inv(\pi_1)$, where Inv represents the number of inversions of a permutation.

Currently, the fastest method for exactly computing the number of inversions of a permutation runs in time $O(n \log n / \log \log n)$ using the subset rank data structure of Dietz (though there is no known superlinear lower bound for the problem). However, finding an approximate number of inversions is much easier. Diaconis and Graham [DG77] showed that

$$Inv(\pi) \leq D(\pi) \leq 2Inv(\pi) \ , \tag{2.3}$$

where $D(\pi) = \sum_{i=0}^{n-1} |\pi(i) - i|$.

The decisions in the binary search are now done as follows. In the first step of the search, the lower half is chosen if $D(\pi_1) < \frac{2}{3}\frac{n^2}{2}$. According to (2.3), this kind of choice guarantees to yield no more than

$$\min\left(D(\pi_1), \ \frac{n^2}{2} - \frac{D(\pi_1)}{2}\right) \leq \frac{2}{3}\frac{n^2}{2} \tag{2.4}$$

bad parameters in the selected interval. Similarly, in later steps the decision to choose the lower half is made if $D(\pi_1)$ is less than 2/3 of the upper bound on the number of bad multipliers in the current interval. The upper bound is determined by an expression analogous to the left-hand side of (2.4), with the corresponding values taken from the previous step. Consequently, $\log_{3/2}\frac{n^2}{2}$ steps are needed to isolate a suitable multiplier. The initial search space must be large enough to allow this number of iterations, so it must be of size at least $(n^2/2)^{1/\log\frac{3}{2}}$. The value of c may be set to $2 \cdot 1.71 = 3.42$.

The technical description of efficient computation of $D(\pi_1)$ is given in Section 2.8.2. The techniques used are standard for algorithms that parallelize operations on word level. The computation of $D(\pi_1)$ is the only non-trivial step performed in the algorithm for finding a suitable multiplier. Assuming that the input to the algorithm for multiplier selection is given in a word-packed form, a bound on the running time is

$$O\left(n \log n \left(\frac{1}{K} + \min\left(\log\log n, \frac{\log n \log K}{K}\right)\right)\right), \tag{2.5}$$

where K is the number of keys that can be packed in a machine word. The expression with the minimum corresponds to the choice of sorting procedure. As before, we have that K can be as high as n. The algorithm needs $\Omega((\log n)^3)$ time even for extremely large words, say $w = 2^n$.

2.4 General Universe Reduction — Top-down Approach

The basic function f can be combined in different ways to achieve a larger reduction of universe. The ultimate goal is to have a function that maps original

keys to signatures of $O(\log n)$ bits. In this section we discuss the most natural approach, direct composition of (several versions of) function f. This method allows the final function to have a rather succinct description on a word RAM. An interesting feature of the method in the practical RAM model is efficient batched evaluation.

We will define a sequence of functions $(g_0, g_1, \ldots)$ that are created through compositions of function f. A set $\{a_0, a_1, \ldots\}$ represents multiplier parameters, and the shrinking ranges of the functions will have bit-lengths given by the elements of the sequence v defined by $v_0 = w$, $v_{k+1} = \lceil v_k/2 \rceil + \lceil c \log n \rceil + 1$. By induction, for $k > 0$ it holds that

$$v_k < \frac{w}{2^k} + (\lceil c \log n \rceil + 2)\left(1 + \frac{1}{2} + \cdots \frac{1}{2^{k-1}}\right) \ .$$

As a result, $v_k < \frac{w}{2^k} + 2\lceil c \log n \rceil + 4$, for all k. Define $g_0(x) = f(x, \lceil v_0/2 \rceil, a_0)$, and for $k > 0$, define $g_k : U \to [2^{v_{k+1}}]$ with $g_k(x) = f(g_{k-1}(x), \lceil v_k/2 \rceil, a_k)$. Let $k_0 = \lceil \log \frac{w}{c \log n} \rceil - 1$. Using one of the selection algorithms from Section 2.3, it is possible to set values of the parameters from $\{a_0, a_1, \ldots, a_{k_0}\}$ so that the function g_{k_0} maps each original key from S to a unique signature of $3\lceil c \log n \rceil + 4$ bits. In other words, k_0 simple steps are needed to perform the universe reduction to a polynomial-size universe.

We state a technical result regarding the evaluation of the function g_{k_0}. The result is a feature of the function family — it is independent of the way the parameters are chosen. The technical proof is given in Section 2.8.1.

Lemma 2.4.1. *Let $k_0 = \lceil \log \frac{w}{c \log n} \rceil - 1$. Given a set of keys $\{x_1, x_2, \ldots, x_{k_0}\}$ from U, the values of all signatures $g_{k_0}(x_i)$, $1 \le i \le k_0$, can be computed in time $O(k_0)$ on a word RAM. On a practical RAM the time is $O(k_0(\log \log n)^{1+o(1)})$.*

The procedure for batch evaluation makes use of word-level parallelism. To obtain the following result it remains to analyze the cost of instantiating the parameters $a_0, a_1, \ldots, a_{k_0}$.

Proposition 2.4.2. *In the word RAM model with word length w, for a given set S of n keys there is a function $h : [2^w] \to [n^{O(1)}]$ that is one-to-one on S, with the following properties: (i) the description of h has a size of $O(\log \frac{w}{\log n} \cdot \log n)$ bits; (ii) the description can be computed in time $O(n \log n (\log \log n)^2 + (\log n)^3 \log w)$ with a deterministic algorithm; (iii) the evaluation time of h on single argument x is $O(\log \frac{w}{\log n})$. Further, for any set $T \subset U$ of $\Omega(\log \frac{w}{\log n})$ arguments, the values $h(x)$, $x \in T$, can be computed in total time $O(|T|)$.*

Proof. For setting of parameter values we decide to use the variant of the selection algorithm that is described in Section 2.3.4. Sorting operations are dominant in execution times, see (2.5). A couple of sorting operations are performed at every bisection step of every binary search. In the first few levels of the reduction, more precisely for $k \le \log \log n$, we choose to utilize Han's sorting algorithm in selection of a_k. In that phase, a total of $O(n \log n (\log \log n)^2)$ time is spent on sorting tasks. After completion of the first phase we have that $K = \Omega(\log n)$ values of type $g_k(x_i)$ can fit in single machine word. To take advantage of this,

for $k > \log \log n$ the algorithm of Albers and Hagerup is used. The total cost of sorting tasks in the second phase is

$$O\Big(\sum_{j=\log \log n}^{k_0} \max \big\{ n(\log n)^2 \frac{j}{2^j}, \ (\log n)^3 \big\} \Big) = O(n \log n \log \log n + \log w (\log n)^3)$$

because of the halving of the key sizes and compactions similar as in the procedure of Lemma 2.4.1 (for implementation details, see the proof of Lemma 2.4.1 in Section 2.8.1). □

It is interesting to mention that other known types of functions suitable for deterministic selection [FKS84, Ram96, HMP01, Ruž08b] require a description of size $\Omega(w)$ bits, when the range is of size $n^{O(1)}$.

To adapt the construction procedure to the practical RAM model we simply replace machine instructions for multiplication with software routines. Multiplying two w-bit numbers requires $(\log w)^{1+o(1)}$ time with the restricted instruction set [BMM97]. We need full-word multiplication mainly in the general sorting procedure in the first $\log \log n$ reduction steps. Outside of sorting procedures we have full-word multiplication with special, simple constants only; it is simpler and more efficient to replace those instructions with combinations of shift and addition instructions. The second phase of the construction procedure, where $k > \log \log n$, requires only block multiplication of "small" numbers of $O(\log n)$ bits (for details, see the proof of Lemma 2.4.1). The result on universe reduction in the practical RAM model is formally stated in Proposition 2.2.5. A number of constants needed to carry out the "software" multiplications are computed during the construction. The term $(\log w)^{O(1)}$ in the construction bound represents time for preparing those constants.

2.4.1 Going down to size $n^{2+\epsilon}$

In the definition of the function sequence g we decided to always set the parameter s of the function f to equal half of the bit-length of the domain of x. Many other choices would do as well, affecting running times of the algorithms by a constant factor. Alternative values of s which do not increase the running times are those which are "a little" smaller than one half of the bit-length. For example, we may set $s = \lceil (w - \lceil c \log n \rceil)/2 \rceil$ for the top-level function, and so on. Then, instead of the sequence v we may define a sequence $\hat{v}$ with $\hat{v}_0 = w$, $\hat{v}_{k+1} = (\hat{v}_k + \lceil c \log n \rceil)/2 + 2$. It holds that $w/2^k < \hat{v}_k < v_k$. The elements of the modified function sequence $\hat{g}$ are $\hat{g}_k : U \to [2^{\hat{v}_{k+1}}]$, $\hat{g}_k(x) = f(\hat{g}_{k-1}(x), \lceil (\hat{v}_k - \lceil c \log n \rceil)/2 \rceil, a_k)$. There is a small advantage of $\hat{g}$ in comparison with g. With g, if we go beyond k_0 the image sizes of g_k converge to $2\lceil c \log n \rceil + O(1)$ bits, which means that the reduced universe cannot be smaller than n^4. On the other hand, the image sizes of $\hat{g}_k$ converge to $\lceil c \log n \rceil + O(1)$ bits. The algorithm from Section 2.3.3, based on exact computations of m_1 values, allows setting c to 2. Consequently, for any fixed $\epsilon > 0$ there is $k_1 = k_0 + O(1)$ and a function $\hat{g}_{k_1}$ with a range of size $n^{2+\epsilon}$. The time taken to construct $\hat{g}_{k_1}$ is $O(n \log n (\log n + \log w))$ on a word RAM. Note that we can also use the function

sequence $\hat{g}$ or a similar one in combination with the faster selection algorithm from Section 2.3.4. In that case the final range can be of size about $n^{3.5}$.

2.5 General Universe Reduction — String Approach

In Section 2.4 we saw one way of combining several versions of the function f that achieves large reduction of the universe. Another approach is to view keys as strings over some chosen alphabet, and then apply f on (some of) the characters, and combine the values in some way. Many concrete variants of this approach can be imagined, yet we will need only two. Let $x[0]x[1]\ldots x[q-1]$ be a string representation of key x over some alphabet. One possibility is to apply f to all characters and concatenate the resulting values, viewed as binary strings. We may use the same multiplier parameter for all characters, and thus the length-reduced value for key x after one level of reduction has a form of $f(x[0],a)f(x[1],a)\ldots f(x[q-1],a)$. The process is repeated with different multipliers and possibly different alphabets at subsequent levels of reduction. We will refer to this way of combining function f as the *parallel reduction*. In the second version, which we call *suffix reduction*, only the last characters get reduced at a single reduction level. Although the structure of reduction sequences is different for those two variants, as well as the processes of parameter selection, the final functions for those two compositions can have similar *dot product* forms. A precondition for this is to suitably set intermediate alphabet sizes in the parallel reduction. For example, after two levels of parallel reduction we want the function to have a form of $x \mapsto f(f(x[0],a)f(x[1],a),a')\ldots f(f(x[q-2],a)f(x[q-1],a),a')$. Having the final functions in dot product form means that they can be evaluated rather efficiently on a word RAM, as we will see in Section 2.5.3.

The method of suffix reduction allows a smaller range of the final function. We may go down to the desired $O(\log n)$ bits with one application of the method. However, the construction time is reasonably fast only for relatively small string-length q. On the other hand, parallel reduction allows rapid construction for large keys, and reasonably fast construction for some intermediate range of key lengths. The result from Theorem 2.2.1 is obtained by having two stages of reduction, one for each method. First, using parallel reduction we map keys of w bits to keys of $O(\log n \log \frac{w}{\log n})$ bits. Suffix reduction then takes over, and maps the given values of size $O(\log n \log \frac{w}{\log n})$ bits to $O(\log n)$-bit values.

The presentation is mainly for the word RAM model. Adapting to the external memory setting with fixed-length keys is trivial. Some issues are nonexistent when only performance of memory transfers is analyzed, and the algorithms become simpler. Variable-length keys are treated in Section 2.6. For the word RAM implementation we assume that $w < n^{O(1)}$.

2.5.1 Parallel reduction

Let $x[i]_\sigma$ denote the ith character of key x viewed as a string over alphabet $[2^\sigma]$. If σ is not too small, e.g. $\sigma > 4\log n$, we may apply f to individual characters

and concatenate the resulting values, viewed as binary strings. The length-reduced value for key x can be written as $f(x[0]_\sigma, s, a) f(x[1]_\sigma, s, a) \ldots f(x[q-1]_\sigma, s, a)$, where $q = \lceil \log u / \sigma \rceil$ and x is zero-padded to make $x[q-1]$ a σ-bit value (if necessary). For integer keys on a word RAM, all q characters fit in one machine word. This way of computing signatures is used in sorting algorithms from [AHNR98, FPP06]. They use different types of hash functions and the functions are chosen randomly. Here we investigate what can be achieved with this approach by using our type of hash functions.

To make the mapping to the reduced universe injective on S it is not necessary to make f injective on all character values that appear in the keys. Observe the trie for S, with the elements of S viewed as strings over alphabet $[2^\sigma]$. It is enough to make f injective on the set of character values that correspond to outgoing edges of branching nodes in the trie. The number of such edges is equal to the number of leaves plus the number of branching nodes minus one. In a trie with n leaves, this is at most $2n - 2$ edges. We will call the characters that are the labels of outgoing edges of branching nodes as *branching characters*.

Three tasks can be identified at each level of reduction: the first one is to collect all branching characters, the second one is to remove possible duplicate values, and the third one is to find a suitable multiplier for the collected set of σ-bit values. We discuss them in reverse order.

The algorithm of Section 2.3.4 takes $O(n(\log n)^2 \frac{\log q}{q})$ time to yield a suitable multiplier. We will soon describe how exactly to set the value of the parameter σ at each level. Generally, the value of σ will not decrease across reduction steps. Beside being non-decreasing, the character lengths will stay within a constant factor from the initial value of σ at the first level of reduction. Consequently, the time spent on the third task does not change much over different levels, and the total time for all the levels is $O(n(\log n)^2 \frac{\log q}{q} \log \frac{w}{\log n})$, where the value of q from the first level is assumed, for concreteness.

The process of removing duplicate values among branching characters can be performed efficiently even when many values are packed in a machine word. We make sure that fields that store the collected characters have size at least $\sigma + 1$. We start with sorting, of course. Doing an XOR operation on each pair of neighbouring values, it is possible to transform each run of equal values to a nonzero value in the first field in the run followed by a sequence of zero fields. Using standard word-parallelism tricks, in a few operations it is possible to zero-out the character values that are aligned with the zero fields, and to leave all the other characters intact. Finally, a sorting operation pushes all the empty fields to one end of the sequence of word-packed characters.

To carry out the task of collecting the branching characters we may perform the first two phases of a known algorithm for constructing a *path-compressed* trie. The first step is to sort the input keys; w.l.o.g. assume that the sorted sequence is $(x_1, x_2, \ldots, x_n)$. In the the second step for each $i < n$ we find the first distinguishing character position for x_i and x_{i+1} (in other words, find the longest common prefix). If such position is j, then $x_i[j]$ and $x_{i+1}[j]$ are added to the set. Locating the first distinguishing character amounts to finding the most significant one-bit in a word, which can be done in constant time [FW93, AHNR98].

It turns out that after the first level of the reduction, the task of collecting the branching characters can be done in a simpler and faster way than with the two phases of the path-compressed trie construction. The topology of the tries (in unordered versions) cannot be preserved across different levels, since we wish to conduct a large reduction of the universe. Yet, the topology changes gradually and in a nice way, especially if we choose alphabet sizes suitably. We do not really need a representation of the trie, only *the set* of labels of outgoing edges of branching nodes (of the full trie), meaning that the order in the sequence of collected characters is not relevant to us.

We initialize σ to $c \log n \log \frac{w}{c \log n}$ at the first step. Let v_1 be the number of bits needed to represent a result of the function f at the first level. Let us choose the character length at the second level to equal $2v_1$. Continuing in this fashion, we set the character length at the $(i+1)$st level to equal $2v_i$, where v_i is the bit-length of the image of f at the ith level. If two strings have a common prefix of $2p$ characters at the beginning of level k, they will have a common prefix of p characters at the beginning of level $k+1$. If the length of the longest common prefix of two strings is $2p+1$ at level k, the longest common prefix at level $k+1$ will be of length p. This process of trie "unfolding" and pushing the branching nodes towards the root is depicted in Figure 2.1. Suppose that at level k the first distinguishing characters for every pair of strings are included in the collected set (such characters for a pair of strings may have got extracted from some other strings with which they share appropriate prefixes). That property continues to hold at level $k+1$ as long as we decide to collect every character that contains an image of a branching character from level k. At every level at most two characters from one string are extracted and added to the set of branching characters. If at the kth level characters at positions $2p$ and $2p+1$ are the selected characters of some string, then starting from level $k+1$ only one character will be taken from that string.

The size of the range of the function f that operates on characters at a reduction step depends on the alphabet size, the size of the domain of multipliers and the value of the parameter s. Deciding on a value for s is a simple matter; we may always set $s = \sigma/2$. The size of the multiplier domain is again $c \log n$ bits. As in the top-down approach, the key length will get "almost" halved after every step. The initial value of σ was set so that the final character length in bits stays only a constant factor larger.

Although the values of keys get properly reduced, their physical representations are not automatically reduced in this variant of the string approach. In the I/O model, compacting the strings with reduced character values is a trivial operation. However, on a word RAM we achieve compaction through interleaving of characters of different strings. Compactions are performed using shift-and-adds, like in the procedure of Lemma 2.4.1, only here we operate at the level of characters, not whole keys. The order of the characters within one key stays the same as in the standard representation. The characters of a string are spread out at equal distances, and they are all inside one word. Different keys in a word appear at different offsets. This representation is illustrated in Figure 2.2. Logical masks used for extracting branching characters can be easily maintained with this representation as well. Masked characters can be collected
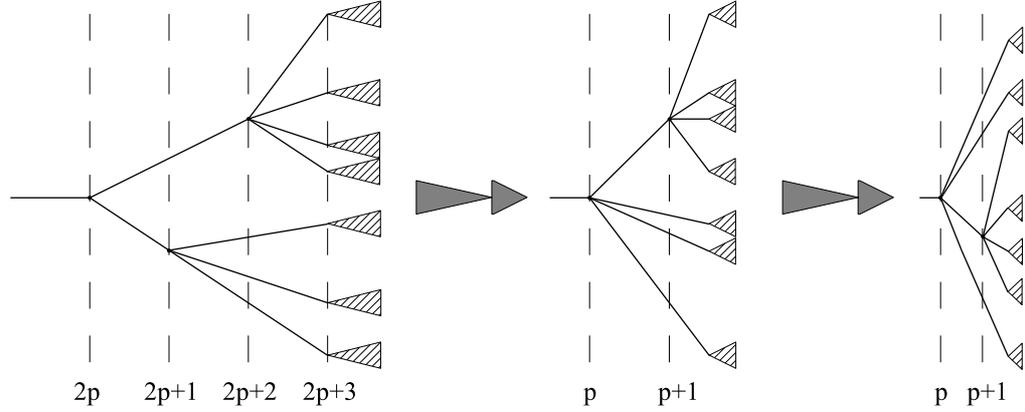
Figure 2.1: Trie unfolding. The leftmost illustration shows a part of the trie at the beginning of the kth level of reduction; the middle one shows the corresponding part at the beginning of level $k + 1$, and the final one represents the end of level $k + 1$.

by pushing them to the highest-order portion of a word using multiplication, as illustrated in Figure 2.3. The interleaved representation of strings is not relevant to the tasks of removing duplicates and finding a suitable multiplier.

By inspecting all parts of the construction procedure and substituting $q = \Theta(w/(\log n \log \frac{w}{\log n}))$, we find that the total running time is

$$O(n\frac{(\log n)^3}{w}(\log \frac{w}{\log n})^3 + n) \ ,$$

assuming that the input sequence of keys is already sorted. Finally, we need to express the structure of a partially reduced key in terms of the original key value. That way we also get the form of the final reduction function, which is important for the evaluation process. Let $\oplus$ denote the concatenation operator. It is not hard to see that after k levels of the reduction, a key $x \in U$ is mapped to

$$\bigoplus_{i=0}^{\frac{q}{2^{k-1}}-1} \sum_{j=0}^{2^k-1} \alpha_j \cdot x[i \cdot 2^k + j]_{\sigma/2} \ , \ \text{where} \tag{2.6}$$

$$\alpha_j = \prod_{l=0}^{k-1} \phi(j, l, a_l) \ , \quad \phi(j, l, z) = \left\{ \begin{array}{ll} z & \text{if } l\text{th bit of } j \text{ is } 1, \\ 1 & \text{otherwise} \end{array} \right. \ .$$

Given parameters a_l, $0 \le l \le O(\log \frac{w}{\log n})$, in time $O(\log w)$ it is possible to compute all the multipliers α_j, $0 \le j < 2q$, that are used in the final reduction function. Here computing all the multipliers α_j means creating a word containing the sequence $(\alpha_0, \alpha_1, \ldots, \alpha_{2q-1})$, which is exactly what we need.
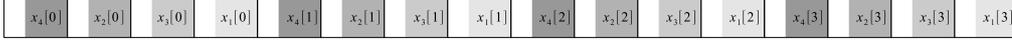
Figure 2.2: Interleaved string representation. The white-spaces represent zeros which complement the characters within fields.
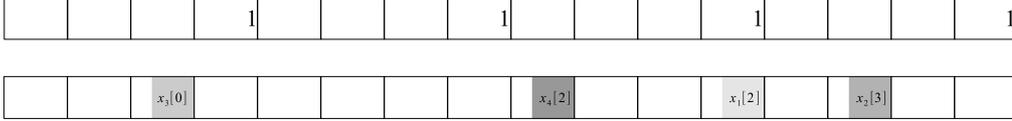
Figure 2.3: Collecting branching characters using multiplication.

2.5.2 Suffix reduction

In the method of suffix reduction we want to find multipliers $a_0, a_1, \ldots, a_{q-2}$ such that the function

$$x \mapsto a_0 \cdot x[0]_\sigma + (\ldots + (a_{q-3} \cdot x[q-3]_\sigma + (a_{q-2} \cdot x[q-2]_\sigma + x[q-1]_\sigma)) \ldots) \quad (2.7)$$

is injective on S. The conceptual sequence of reductions is designated by the parentheses in the expression. A partially reduced key after k reduction levels consists of the prefix $x[0]_\sigma \ldots x[q-k-2]_\sigma$ of the original key x along with a reduced value of the remaining suffix $x[q-k-1]_\sigma \ldots x[q-1]_\sigma$. The process of parameter selection proceeds in accordance with the reduction sequence, starting from a_{q-2}, and finishing with a_0.

In this variant of the string approach we may set $\sigma = O(\log n)$. The range of the final function has a size of $O(\log n + \log q) = O(\log n + \log \frac{\log u}{\log n})$ bits. Therefore, the final range has a size of $O(\log n)$ bits, except for very large key lengths. In our framework we mainly use the suffix reduction method for the second phase of the general reduction. The input universe for the second phase has a size of $O(\log n \log \frac{w}{\log n})$ bits, which is relatively small.

Beside the calls to the multiplier selection procedure, in the construction algorithm there is a preparation task of splitting and partitioning the keys so that the elements in the sets $\{x_1[0], x_2[0], \ldots, x_n[0]\}$, $\{x_1[1], \ldots, x_n[1]\}$, $\ldots$, $\{x_1[q-1], \ldots, x_n[q-1]\}$ are grouped together. The cost of this partitioning is proportional to $\log q$ times the cost of reading the keys (they may be packed into words). However, the computationally dominant task is multiplier selection. The selection algorithm is applied $q-1$ times. From (2.5) we see that the cost of one call to the selection procedure is $O(n \frac{(\log n)^3}{w} \log \frac{w}{\log n} + (\log n)^3)$, since $K = \min\{\Theta(\frac{w}{\log n}), n\}$ input elements can fit in one word. Because $q = O(\log \frac{w}{\log n})$, the total running time of the construction algorithm is $O(n + n \frac{(\log n)^3}{w}(\log \frac{w}{\log n})^2)$.

When evaluating the function from (2.7) we obviously do not need to go through the reduction sequence from the construction process. The function can be evaluated instantly using a dot product calculation.

2.5.3 Function evaluation

On a word RAM, dot product of integer vectors packed into single words can be computed in constant time when one operand is a constant and the word size is not very large relative to n. The constant operand can be preprocessed offline so that dot products can later be computed via standard multiplication. Denote the arguments by x and a, where the value of a stays constant. The vectors have d integer components of size $\Omega(\log n)$ bits. The procedure needs to calculate $\sum_{i=0}^{d-1} x[i]a[i]$. The vector x is represented in dense word-packed form; in other words, it is just a binary string from $\{0,1\}^w$ interpreted in different way, like in the previous subsections. Let σ be the bit-length of components of x and a. For simplicity, we will suppose that the values of the components of a can be represented using at most $\sigma - \log w$ bits, as this typically is the case in our applications of the procedure. The top $\log w$ bits in every field of a are assumed to be zero (if this could not be fulfilled, it would not be a problem to make additional splits of the operands, as long as $w \leq n^{O(1)}$).

Let a^R be the vector produced from a by reversing the order of the fields, that is $a^R[d-1-i] = a[i]$. We split a^R into two parts, a' and a'', such that the fields of a' and a'' alternate between zero fields and fields retained from a^R. We have that $a^R = a' + a''$. The vectors a' and a'' are prepared at the end of the construction procedure, when constant a is found. We need to preprocess one constant for the parallel reduction and one for the suffix reduction.

During evaluation the vector x is split into two parts, call them x' and x'', the same way the vector a^R was split. Suppose first that d is even. The dot product value is given by

$$((x' \cdot a'' + x'' \cdot a') \; \texttt{SHR} \; (d-1)\sigma) \; \texttt{AND} \; (2^\sigma - 1) \; ,$$

where $\texttt{SHR}$ stands for the operation of right shift, and the multiplication operators denote standard multiplication of w-bit integers. Splitting of operands into two parts prevented overflows from spoiling the result. If d is odd, the resulting value is given by $((x' \cdot a' + x'' \cdot a'') \; \texttt{SHR} \; (d-1)\sigma) \; \texttt{AND} \; (2^\sigma - 1)$.

2.6 Variable-length Keys

It is of interest to discuss making signatures of "real" strings that can span multiple words. We consider binary strings, and they can be of different lengths. Because strings can possibly be long, we also care about I/O performance of the evaluation and construction procedures. Every key is represented with an encoding of the key length followed by a sequence of data bits. Usually, a fixed number of bits can be reserved for storing the length attribute. In general, a prefix-free code is sufficient. If it can be assumed that all length codes have the same size, the method becomes a bit simpler. We will discuss the general case. No distinction is made between the length part and the data part of a string in this general variant of the method, they are reduced together as one key.

The reduction again starts with a process of parallel reduction. Most of the points from Section 2.5.1 translate easily to this setting. However the parameter w, which figured in the setting of alphabet sizes, does not have a counterpart

in the case of unbounded and variable length strings. Here we choose to set the initial character length to $c \log n \log \log n$ bits. We utilize the same construction procedure for $\log \log n$ levels of reduction. If at some level a key is comprised of only one character then it is set aside to enter the second phase of the reduction; keys set aside have lengths of $O(\log n \log \log n)$ bits. After $\log \log n$ levels, all the remaining keys have lengths smaller than their original lengths by a factor of $\Omega(\log n)$. At that time, a full sorting operation can be afforded. The process is then repeated by resetting the character length to $c \log n \log \log n$.

In the second phase, signatures of length $O(\log n \log \log n)$ bits are further reduced by suffix reduction, or the top-down composition from Section 2.4 if only I/O performance is relevant. However, elements that completed the first phase at different times may have equal signatures. For that reason, we group elements according to time they take to finish the first phase of the reduction. Let S_k be the set of elements of S that were reduced to one character after k levels. Every nonempty set S_k is assigned a unique identifier from $[n]$. The final signature value of a key from S_k is composed of the assigned identifier of the set S_k and the key's signature produced by the second-phase function. The final signatures are unique over S, and they have a length of $O(\log n)$ bits. To make all input values for the second phase have the same string-length, we implicitly pad with zeros all keys from S_0 to length $c \log n \log \log n$ (the elements of S_0 originally have smaller lengths).

If L is the maximum length of strings in S, there are $O(\log \frac{L}{\log n} + \log \log n) = O(\log L)$ levels of reduction in total. For every level, storage of the multiplier parameter occupies $c \log n$ bits. There are also $O(\log \frac{L}{\log n})$ distinguished groups of keys, some of which may be empty. Associating the number k with the identifier of (nonempty set) S_k requires $O(\log n)$ bits of storage. In total, the description of the entire signature function fits in $O(\log L \log n)$ bits.

The result for the cache-oblivious model is stated in Theorem 2.2.7. It is not hard to check that the expression in (2.1) covers both phases of the construction. A similar result for strings in the word RAM model could also be stated. An important adjustment is making a phase transition when reduced values become smaller than the word size. This ensures efficient evaluation on a RAM. Also, some processes are simpler when values span more than one word. For example, there are no interleavings of physical representations of keys (see Section 2.5.1). For any key we may freely spend $\Omega(1)$ time per reduction level, as long as the reduced value does not become smaller than the word size. Keys that are originally short, smaller than one word, are implicitly zero-padded to word length.

2.7 Predecessor Structures

We now turn to the problem of answering predecessor queries on sets of unbounded and variable-length binary strings. Basically the same structure lies behind Theorem 2.2.3, which is for the word RAM model, and Theorem 2.2.9, which is for the external memory models. Differences are in the sorting procedure that is called by the construction algorithms, and in dictionary substruc-

tures. Sorting needs to be optimized for one of the computational models. For both models we can use variants of merge sort. Dictionary substructures that we use are a composition of the signature functions from this chapter and the dictionary for polynomial-size universes from Chapter 3. That combination is sufficiently fast for this purpose because polylogarithmic factors in construction times can easily be absorbed. For concreteness, let the further discussion refer to the structure that is analyzed in the I/O model (Theorem 2.2.9).

Our predecessor structure for strings has two pieces. The top piece is used to find a "small" neighbourhood of the result, and the bottom piece is an array of simple comparison-based structures built over those small neighbourhoods. This division technique is commonly used to absorb or amortize a higher operation cost on some part of a data structure. In our case, it absorbs higher construction cost of the top piece.

To begin the construction, the sorted input is partitioned into groups that have size of $\log^3 n$ elements each. A key (string) of minimum length from each group is included into the input to the top structure. In the bottom array, binary search is used as the search method; additional information about longest common prefixes is needed, e.g. as in a known method for searching within suffix array. Search results can be tabulated for all strings of length smaller than $\log n$ bits. Doing that, for very small strings we avoid paying a cost of $\Theta(\log \log n)$ I/Os for searching within the bottom structures.

The method behind the top piece is a variation of the method of the vEB structure [vEBKZ77]; we made it suitable for variable-length keys, keeping the construction I/O efficient. On a conceptual level it may be said that we use dictionary structures for jumping between the edges and nodes of the trie for the set of keys viewed as binary strings.

The top piece in the predecessor structure

The actual structure will not store a full representation of the trie; we avoid trie traversals in order to get a simpler I/O-efficient construction algorithm. Both the search procedure and the construction procedure of our structure can be thought of as recursive procedures, with every step of recursion reducing bit-lengths of keys by a constant factor. We will describe one step of each these procedures. We observe the full, not path-compressed, trie for the input binary strings.

Two pointers to bottom structures are associated with every branching node, call them *pmin* and *pmax*. The value of *pmin* for a node t indicates the position in the bottom piece of the minimum element in the subtrie rooted at t. Analogously, *pmax* points to the maximum element in the subtrie rooted at t.

Suppose that the search procedure is to find the predecessor of string x in the subtrie rooted at node u, where u need not be a branching node. If u is not a branching node, equality tests involving a prefix of x can lead us to the first branching node. If there is a mismatch with the prefix of x then we easily resolve the query — either the predecessor of x is the maximum element in the subtrie, or the successor of x is the minimum element of the subtrie. The ultimate result is found in the bottom piece, indexed by either *pmin* or *pmax*. If x matches

until the first branching node, then we check whether the branch that should be followed leads to a single leaf. If yes then we easily handle the case; otherwise we will use signatures to skip "big" parts of the trie. Remark that the branch that should be followed is decided by the value of a single bit.

Let t denote the first branching node in the trie rooted at u, and let x_t be the string obtained by discarding the prefix of x that led to t. W.l.o.g. we assume that x_t is not longer than the height of the subtrie rooted at t; any further bits would be insignificant to the search procedure. Let l be the power of two that is nearest to $|x_t|/2$. For every power of two smaller than the height of the subtrie there is an associated string dictionary. We look up the value of l-bit prefix of x_t in the dictionary associated to l. If there is a match, we retrieve the node u' up to which x certainly matches. The search continues in the subtrie rooted at u' with the remaining suffix of x_t (and hence x) as the argument. Otherwise if there is no match, the search proceeds in the upper part of the trie rooted at t, with the l-bit prefix of x_t as the argument. This means that a dictionary lookup with the $l/2$-bit prefix of x_t follows, and so on. Each edge of the trie is "cut" at most once, and only one dictionary may hold an entry that refers to the edge.

Construction of the structure. Initially we find the first branching node of the entire trie for S, and store information about the prefix that leads to that node. Now consider the strings with the initial prefix removed and suppose that the trie starts with a branching node. Let l be the height of the trie, rounded up to the nearest power of two. For every $k > 0$, strings having length between $l/2^{k+1}$ and $l/2^k$ are put in a separate set that will "join" the upper-part recursion of the construction after k steps; here we sort the strings according to their lengths. To get an effect of cut, the $l/2$-bit prefixes of the remaining strings are gathered in sorted order; no sorting procedure needs to be executed since we assumed that the input sequence of strings is given in sorted order, and it is trivial to maintain that order in this algorithm. All values of $l/2$-bit prefixes that appear only once are put aside into the set for the upper-part recursion; keys corresponding to such values can be distinguished in the upper part of the trie. For every maximal run of strings with equal values of $l/2$-bit prefixes, the prefix value is put in the dictionary and with it is associated a pointer to the structure that will be built in the lower-part recursion over the set of suffixes of the strings in the run. The value of the prefix enters the set for the upper-part recursion. Initializing pointers *pmin* and *pmax* at branching nodes is a trivial matter.

2.8   Appendix: Technical Details

2.8.1   The proof of Lemma 2.4.1

We describe the computational procedure for the word RAM model, and afterwards we state small adjustments for the practical RAM model. Let $[\![x]\!]$ denote $2^{\lceil \log x \rceil}$, that is, the smallest power of two that is not smaller than number x. W.l.o.g. we assume that w is a power of two; if this is not the case we may use the largest portion of the word whose size is a power of two.

The result of function g_{k-1} can be represented with v_k bits. It serves as the argument of function $f(x, \lceil v_k/2 \rceil, a_k)$ in the computation of g_k, and so on. We store the values $g_{k-1}(x_i)$ in fields of $\lceil\!\lceil v_k \rceil\!\rceil$ bits. Fields are packed in words, with $w/\lceil\!\lceil v_k \rceil\!\rceil$ fields in each word, except that in the last word some fields may be empty. Within each field the contained number is stored right justified, meaning that the least significant bit position of the number is aligned with the least significant position of the field. Possible empty space of $\lceil\!\lceil v_k \rceil\!\rceil - v_k$ bits is filled with zeros.

The fields holding the values from $(g_{k-1}(x_i))_{i=1}^{k_0}$ will not appear in order of increasing i. As k increases they will get more and more shuffled. Once the values from $(g_{k_0}(x_i))_{i=1}^{k_0}$ are computed, the original order will be restored.

It can be seen that $v_k < \frac{1}{2}v_{k-2}$, for $k \leq k_0 - 1$. Hence, while $k \leq k_0 - 1$ the field size decreases by a factor of two at least once in two levels. When k is lower (and v_k larger) the shrinking happens more frequently.

We state the process of computing the values $g_k(x_i)$ given the values $g_{k-1}(x_i)$ in the word-packed form specified before. Let $l = w/\lceil\!\lceil v_k \rceil\!\rceil$. There are l fields in every word and they start at bit positions $i \cdot \lceil\!\lceil v_k \rceil\!\rceil$, $0 \leq i < l$. When step k starts, the constant

$$M_l = \sum_{i=0}^{l-1} 2^{iw/l}$$

will be available as part of the output of step $k - 1$. The number M_l has one-bits exactly at starting positions of fields within a word. Let `SHL` stand for the operation of left shift, and `SHR` stand for the operation of right shift. The following preprocessing stage is done for level k:

```
phi_offset ← ⌈v_k/2⌉
mask_psi ← ((1 SHL phi_offset) - 1) · M_l
mask_phi ← (-1) - mask_psi
```

Later, in the discussion for the practical RAM model, we will see that the multiplication by M_l can be avoided for free. The procedure executed on a word X is:

```
Z ← X AND mask_psi
Y ← (X AND mask_phi) SHR phi_offset
X ← Y + a_k·Z
```

The procedure is repeated on all words which contain fields with $g_{k-1}(x_i)$ values.

If $\lceil\!\lceil v_{k+1} \rceil\!\rceil < \lceil\!\lceil v_k \rceil\!\rceil$ the fields are compacted as follows. All the words containing the data are partitioned into pairs, except for one word if the number of words is odd. Let X_1 and X_2 be the words in a pair. They are replaced with one word containing $X_1 + (X_2 \text{ SHL } \lceil\!\lceil v_{k+1} \rceil\!\rceil)$. It doesn't matter that one word may be unpaired; some fields will simply be empty. The constant M_{2l} is obtained as $M_l + (M_l \text{ SHL } \frac{w}{2l})$.

At the end of level k_0, when the values from $(g_{k_0}(x_i))_{i=1}^{k_0}$ are computed, all the compaction operations are reversed, thereby putting the signatures into proper order. Entire procedure of computing signatures involves a time of $O(k_0 \cdot (1 + 1/2 + 1/4 + \ldots)) = O(k_0)$.

To adapt the algorithm for the practical RAM model, we need to get rid of multiplications. Observe the level preprocessing procedure. We can actually obtain the mask `mask_psi` in a similar way we obtained M_l using the value for the previous level. Let $\mathtt{mask\_psi}_k$ denote the mask used at level k, and let t_k be

$$\mathtt{mask\_psi}_{k-1} \text{ AND } (\mathtt{mask\_psi}_{k-1} \text{ SHR } (\lceil v_{k-1}/2 \rceil - \lceil v_k/2 \rceil)) \ .$$

The mask t_k has the same number of blocks of 1-bits as $\mathtt{mask\_psi}_{k-1}$, but each block is smaller. If compaction was not performed at level $k-1$ then we set $\mathtt{mask\_psi}_k = t_k$; otherwise set $\mathtt{mask\_psi}_k = t_k + (t_k \text{ SHL } \lceil\!\lceil v_k \rceil\!\rceil)$.

The multiplication by a_k values cannot be avoided. In this place we use the observation of Brodnik et al. [BMM97] on doing multiplication using basic instructions. We need a number of constants precomputed at the time when the data structure is built. To exploit the fact that one multiplier is "small" we use block multiplication. What we want to perform is multiplication of numbers of bit-length $\lceil\!\lceil v_k \rceil\!\rceil/2$ in fields of the word denoted by Z (in the code fragment) and the numbers in fields of the precomputed constant $a_k \cdot M_l$. Yet this is implemented through multiplications of blocks of size $O(\log n)$ bits (two multiplications on split operands followed by an addition). Every field can accommodate the result of the multiplication, since involved numbers occupy one half of field size. The multiplication takes time $(\log \log n)^{1+o(1)}$. Over the whole procedure, we require access to k_0 constants of type $a_k \cdot M_l$, and $k_0 \cdot (\log \log n)^{1+o(1)}$ constants needed by the multiplication routine. Computing all of them takes time $O(k_0^2 + k_0 (\log \log n)^{O(1)})$, which should be added to the structure's construction time.

2.8.2 Computing $D(\pi_1)$

Here we describe the main computational process of the algorithm from Section 2.3.4. Since some operations are similar to those made in the procedure of Lemma 2.4.1, the presentation will not be overly detailed. The structures which are to be packed into words consist of two fields plus one reserved bit at the most significant position of the structure; this bit is used by the sorting algorithm of Albers and Hagerup. The low order field of structure instance contains a key value — in the first step of the binary search the keys are the values x_i, and in the subsequent steps the keys are modified according to the reduction described in Section 2.3.2. The high order field of structure instance accommodates a function value of type $f(x, \mu)$, where x is the key in the second field. The function values are computed at every step of the binary search, as μ takes different values. Let l denote the number of structure instances which are packed into a single word.

Calculating the function values and placing them next to the arguments is done analogously to the method of Lemma 2.4.1, which is described in the previous section of the appendix. Next comes sorting of the packed structures which naturally determines the order $\prec_f$. The function value in each structure needs to be replaced by the $rank_f$ value. Such operation is performed in constant time per word by utilizing the constant $M_l^2 + i \cdot M_l$, where i is the position of the word in the sorted order. To make the second sorting step order the

structures according to $\prec_\phi$, we need to exchange the order of the fields within every structure, which is easy to accomplish. After the second sorting we are ready to calculate $D(\pi_1)$. W.l.o.g. we describe the operations performed on the first word in the sorted sequence. Let P be the word obtained by masking out all the bits except those that belong to the written $rank_f$ values. Because of the exchange of field order, $rank_f$ values are now aligned with the beginning of every structure. The following sequence of operations is executed:

```
D ← (P + (M_l SHL ⌈log n⌉)) - M_l²
mask_positive ← D AND (M_l SHL ⌈log n⌉)
mask_positive ← mask_positive - (mask_positive SHR ⌈log n⌉)
D_p ← D AND mask_positive
mask_negative ← NOT mask_positive
D_n ← (M_l² AND mask_negative) - (P AND mask_negative)
D ← D_p + D_n
return D·M_l SHR (l-1) w/l
```

For the ith word in the sequence, the constant M_l^2 is replaced with $M_l^2 + i \cdot M_l$. The returned values are summed to ultimately produce $D(\pi_1)$. After inspecting the structure of the whole procedure, the upper bound on the running time given in Section 2.3.4 easily follows.

2.8.3 Faster function evaluation for large w

In Section 2.5.3 we saw a way of evaluating the signature functions based on dot-product computation. In this section we describe an alternative evaluation procedure that runs in time $O(\log \log \frac{w}{\log n})$. The running time is nonconstant; yet there is no assumption on the size of w (earlier, for fast evaluation we assumed that $w \leq n^{O(1)}$).

Recall the function representation given in (2.6). When w is very large we cannot compute the final result in constant time on a word RAM, but we can gradually increase the number of levels for which the result can be computed in constant time. We represent (partly reduced) keys in a sparse form, distributing characters uniformly across the word and leaving zeros between the characters. There will be *big* and *small* reduction steps. Conducting each big step requires the same amount of computation, and also conducting each small step requires the same amount of computation. In big steps several levels of application of the function f are carried out at once, while small steps carry out only one level of application of the function f. Between every two neighbouring big steps there is a small step, whose purpose is to ensure that the required conditions hold at the beginning of the next big step. We require the following conditions at the beginning of the mth big step: the string is in the sparse form with characters of σ bits and spaces of $2^{2^m+1}\sigma$ zero-bits between the characters, where σ is the character size at the first level. The mth big step computes the result of 2^m levels of application of function f. The rule of setting the character lengths to values $2v_i$ should hold for levels within each big step. Before the first big step, three single-level reductions are performed to create enough zero-space.

In the mth big step $k = 2^m$ levels are computed in a combined way. The procedure computes all the products $\alpha_j x[i]$, zeros out the unwanted ones, and

then performs appropriate summations. The computation of the products is illustrated in Figure 2.4. The first (constant) multiplication operand and the logical masks can be computed in time $O(\log w)$ during the construction of the functions. After zeroing-out the unwanted products, the summing of terms within groups of 2^k neighbouring terms produces characters of $\sigma + k \cdot c \log n + k$ bits, with zeros in between. In the small step that follows, we carry out one reduction step with function f to compact the obtained characters of $\sigma + kc \log n + k$ bits to characters of σ bits. This is possible with just one application of f because $k < \frac{1}{2} \log \frac{w}{c \log n}$, and therefore $\sigma > \frac{1}{2}(\sigma + kc \log n + k) + c \log n + 1$. The results of the function f may be padded with zeros to get characters of exactly σ bits. There are now more that $2^{2k+1}\sigma = 2^{2^{m+1}+1}\sigma$ zero-bits between characters. All the required conditions are maintained. Consequently, universe reduction to a range of size $O(\log n \log \frac{w}{\log n})$ bits can be done in time $O(\log \log \frac{w}{\log n})$. Going down to a polynomial-size universe takes up $O(\log \log \frac{w}{\log n})$ time with the level-by-level computation.
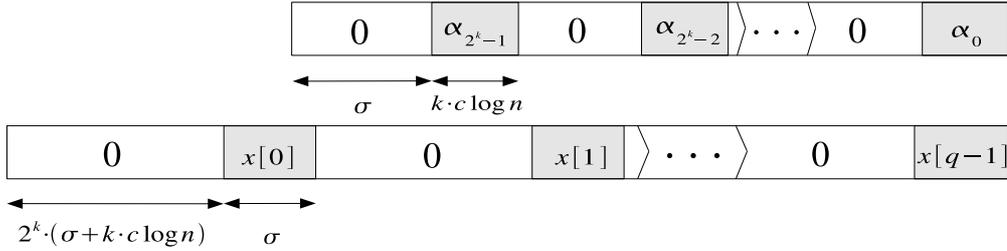
Figure 2.4: Operands in the multiplication that realizes all products $\alpha_j x[i]$.

Acknowledgments

Chapter 3

Constructing Efficient Dictionaries in Close to Sorting Time

Abstract

The dictionary problem is among the oldest problems in computer science. Yet our understanding of the complexity of the dictionary problem in realistic models of computation has been far from complete. Designing highly efficient dictionaries without resorting to use of randomness appeared to be a particularly challenging task. We present solutions to the static dictionary problem that significantly improve the previously known upper bounds and bring them close to obvious lower bounds. Our dictionaries have a constant lookup cost and use linear space, which was known to be possible, but the worst-case cost of construction of the structures is proportional to only $\log \log n$ times the cost of sorting the input. Our claimed performance bounds are obtained in the word RAM model and in the external memory models; only the involved sorting procedures in the algorithms need to be changed between the models.

3.1   Introduction

Dictionaries are among the most fundamental data structures. A dictionary stores a set S which may be any subset of *universe U*, and it answers membership queries of type "Is x in S?", for any $x \in U$. The elements of S may be accompanied by *satellite data* which can be retrieved in case $x \in S$. The size of the set S is standardly denoted by n.

We consider universes whose elements can be viewed as integers or binary strings. In this chapter we concentrate on *static* dictionaries — a static dictionary is constructed over a given set S that remains fixed. *Dynamic* dictionaries allow further updates of S through insertions and deletions of elements. Even static dictionaries are sometimes used as stand-alone structures, but more often they appear as components of other algorithms and data structures, including dynamic dictionaries.

The dictionary problem has been well studied. Many solutions have been given, having different characteristics regarding space usage, time bounds, model of computation, and universe in question. A challenge is to simultaneously achieve good performance on all the terms. We consider only dictionaries with realistic space usage of $O(n)$ registers of size $\Theta(\log|U|)$ bits. In the usual case when $|U|$ is at least polynomially larger than n, this amount of space is necessary (ignoring constant factors) regardless of presence of satellite data. Algorithms involved in construction of a dictionary may be randomized — they require a source of random bits and their time bounds are either *expectations* or hold with high probability. Randomized dictionaries reached a stage of high development and theoretically there is little left to be improved. On the other hand, the progress on deterministic dictionaries was much slower. While in the dynamic case we have some reason to believe that there is a considerable gap between attainable worst-case performance for deterministic dictionaries and the attainable expected performance for randomized dictionaries, there is not any evidence of a required gap in the static case.

A theoretical interest in deterministic dictionaries comes from the question of what resources are necessary to implement an efficient dictionary structure, and random bits are a resource. Having guaranteed time bounds, deterministic structures can be used in systems with strict performance demands. A sufficiently simple deterministic dictionary having comparable performance to a randomized dictionary would make the randomized structure obsolete. Unfortunately, the new solutions described here are not simple enough to be competitive in practice, except possibly in some special cases.

In this chapter we focus on dictionaries with constant lookup time. Because of faster construction time, dictionaries with slightly slower lookups may sometimes be of interest. For example, a structure supporting searches in time $O(\log\log n)$ can be built in linear time on sorted input (Chapter 2).

3.1.1   The word RAM model and related work

The word RAM is a common computational model in data structures literature. It has the machine word size of w bits and a standard instruction set, resembling the primitive instructions of the language C. Execution of any instruction takes one unit of time. A usual assumption for RAM dictionaries is that the elements of U fit in one machine word.[1] Contents of a word may be interpreted either as an integer from $\{0,\ldots,2^w-1\}$ or as a bit string from $\{0,1\}^w$. For more information see, e.g., [Hag98b].

We will list some important results for deterministic dictionaries with constant query time. Each of those results required a different idea, and a new insight into properties and possibilities of some family of hash functions. A seminal work by Fredman, Komlós, and Szemerédi [FKS84] showed that in the static case it is possible to construct a linear space dictionary with a constant lookup time for arbitrary word sizes (no assumptions about relative values of w and n). This dictionary implementation is known as the *FKS scheme*. Besides

---

[1]This assumption simplifies analysis. Some schemes, including ours, scale well when keys are multi-word strings.

a randomized version with expected $O(n)$ construction time, they also gave a deterministic construction algorithm with a running time of $O(n^3 w)$. A bottleneck was choosing of appropriate hash functions. Any *universal* family of hash functions [CW79] contains functions suitable for use in the FKS scheme. Raman [Ram96] devised a deterministic algorithm for finding good functions from a certain universal family running in time $O(n^2 w)$; this implies the same time bound for construction of the FKS dictionary. For $w = n^{\Omega(1)}$, an efficient static dictionary can be built in time $O(n)$ on a sorted sequence of keys. This follows from a generalization of the fusion trees of Fredman and Willard [FW93], and it was observed by Hagerup [Hag98b]. The previously fastest deterministic dictionary with constant lookup time is a result of Hagerup, Miltersen and Pagh [HMP01]. Their construction method has a running time of $O(n \log n)$. There exists an issue with compile-time computation of a special constant that is required for each w, because the only known computation method is a brute-force search that takes time $2^{\Omega(w)}$.

Allowing randomization, the FKS scheme can be dynamized to support updates in amortized expected constant time [DKM+94]. The lower bound result in the same paper states that a deterministic dynamic dictionary, based on *pure* hashing schemes, with worst-case lookup time of $t(n)$ must have amortized insertion time of $\Omega(t(n) \cdot n^{1/t(n)})$ (this lower bound does not hold in general, e.g. see the result of Pagh [Pag00]). A standard dynamization technique [OvL81] applied to the static dictionary from [HMP01] yields a similar type of trade-offs: lookups in time $O(t(n))$, insertions in time $O(n^{1/t(n)})$, and deletions in time $O(\log n)$, where t is a "reasonable" parameter function. The method in Chapter 4 was devised as an alternative to the method from [HMP01] that eliminates the problem with the high compile-time demand.

3.1.2 External memory models

Real computers do not have one plain level of memory but a memory hierarchy. The theoretical I/O-model was introduced to model behavior of algorithms in such a setting, focusing on the cost of I/O communication between two levels of memory [AV88]. Related to the I/O-model is the cache-oblivious model [FLPR99], where the algorithm does not know the size of the internal memory M and the block size B. In these models, comparison-based sorting of n integers (which occupy one memory cell) takes $\Theta(\mathrm{Sort}(n))$ I/Os, where $\mathrm{Sort}(n) = \frac{n}{B} \log_{M/B} \frac{n}{B}$.

From the structures mentioned previously for the word RAM model, it can be observed that the methods from [Ram96] and Chapter 4 can easily be adapted to the external memory models and attain analogous bounds; respectively they are $O(\frac{n^2}{B} \log |U|)$ and $O(\frac{n^{1+\epsilon}}{B})$ I/Os. We take the block size parameter B to represent the number of $\log |U|$-bit items that can fit in a memory block. For the dictionary from [HMP01], no better bound than $O(n \log n)$ I/Os can be stated. The main problem for I/O performance was the dictionary for universes of size polynomial in n, which is a component of the construction from [HMP01].

3.1.3 Background of our techniques

Our contribution consists of two parts. One part is a very efficient dictionary for universes of size $n^{O(1)}$. Beside its use in composition with methods that perform *universe reduction*, this case has a significance of its own. The most prominent example of stand-alone use of dictionaries for "small" universes is representation of a graph. In the problem of storing and (random) accessing edges of a graph, the universe is of quadratic size. The problem is also of interest to some situations in practice, since in reality integer keys are not often very large relative to n. The main part of our structure uses the same kind of hash functions that were used in [HMP01] for this case. Interestingly, those functions are very similar to the functions from [TY79] where construction time was $\Theta(n^2)$. The construction algorithm from [HMP01] runs in time $\Theta(n \log n)$. We devised a different and more efficient construction algorithm.

The other part of the contribution is a follow-up on our technique of making deterministic signatures from Chapter 2. There we introduced a new type of hash functions and associated algorithms for injectively mapping a given set of keys to a set of signatures of $O(\log n)$ bits. The methods are computationally efficient in various models of computation, especially for keys of medium to large lengths. More precisely, when given keys have a length of at least $\log^{3+\epsilon} n$ bits, the algorithms for selecting perfect hash functions have a linear running cost on sorted input. Those functions have rather succinct descriptions, and they might have an application outside of dictionary structures. In our quest for a faster construction in the case $w = \log^{O(1)} n$ we will give up the requirement of complete injectiveness, and replace it with considerably weaker and rather specific properties. These weaker functions will be meaningful only within our dictionary construction.

3.1.4 Our results

The result for the case of universes of polynomial size is summarized in the following theorem.

Theorem 3.1.1. *Let S be any given set of n integers from the universe $\{0, 1, 2,$ $\ldots, n^{O(1)}\}$. In the word RAM model, in time $O(n \log \log n)$ it is possible to deterministically construct a static dictionary over S that performs lookups in constant time and occupies linear space. In the cache-oblivious model, and hence in the I/O model as well, a similar structure can be built using $O(\mathrm{Sort}(n) \log \log n)$ I/Os.*

The method is discussed in Section 3.2. The given structure complements additional results from Chapter 2 in the external memory setting, such as a static predecessor structure for variable and unbounded length binary strings.

In the second part of the paper (Section 3.3) we describe the structures and associated procedures that are efficient when $w = \log^{O(1)} n$. In conjunction with the earlier results, this implies the claimed results for the general case, which are formally expressed in the following theorems. In the performance bounds we plugged in the currently known upper bounds on sorting (which may be optimal).

Theorem 3.1.2. *In the cache-oblivious model, a static linear space dictionary on a set of n keys can be deterministically constructed using $O(\text{Sort}(n) \log \log n)$ I/Os, so that lookups to the dictionary take $O(1)$ I/Os.*

Theorem 3.1.3. *In the word RAM model, a static linear space dictionary on a set of n keys can be deterministically constructed in time $O(n(\log \log n)^2)$, so that lookups to the dictionary take time $O(1)$.*

We could have also listed results for strings, etc. The stated general bounds do not match the actual times in every case. We make remarks on some meaningful special cases, when performance is better.

Remark 3.1.4. *Suppose that $\log |U| = \Omega(\log n \log \log n)$. The construction cost of the dictionary referred to in Theorem 3.1.2 is $O(Sort(n))$ I/Os.*

Remark 3.1.5. *Supposing that $w = O(\log n \log \log n)$, the time taken to build the dictionary from Theorem 3.1.3 is $O(n \log \log n)$.*

Remark 3.1.6. *If $w > \log^{3+\epsilon} n$ and the input set of keys is sorted, the time taken to build the dictionary from Theorem 3.1.3 is $O(n)$.*

At the moment, our fast static dictionaries do not yield an improvement for dynamic deterministic dictionaries. It is one of major challenges in data structures research to either significantly improve performance of dynamic dictionaries, or to prove general lower bounds that would definitely establish a gap between deterministic and randomized dictionaries. How far deterministic dictionaries can go remains unknown, even in the static case.

3.2 Universes of Polynomial Size

3.2.1 Notation and comments

We use the symbol $\oplus$ to denote bitwise exclusive or operation. The number of *collisions* of a function h on a subset A of its domain represents the value $|\{ \{x, y\} : h(x) = h(y) \wedge x, y \in A \wedge x \neq y\}|$. For multisets A and B, the value $|\{ \{x, y\} \in A \times B : x = y\}|$, which may be thought of as the number of collisions between the multisets, is denoted by $\text{coll}(A, B)$. For a multiset A, $A \oplus y$ stands for the multiset $\{x \oplus y\}_{x \in A}$. We use notation $[x]$ to represent the set $\{0, 1, \ldots, x - 1\}$. Also, $\log x$ means $\log_2 x$.

Throughout the presentation, statements of performance bounds for both the word RAM model and the external memory models will appear at several places. The discussion was not separated for different models because we end up with essentially one construction algorithm for all the models, with the only difference being the sorting procedure that gets called (although on a RAM sorting can be avoided at some places, and we may have slightly simpler algorithms). By changing the procedure for sorting, we get methods efficient in the word RAM model, the I/O model, or the cache-oblivious model. In the external memory models we take the block size parameter B to represent the number of $(\log |U|)$-bit items that can fit in a memory block. For this problem, $\log |U| = O(\log n)$.

3.2.2 About universe size

Suppose that the universe has size $2^{\lceil 2 \log N - 2 \log \log N \rceil}$, for some N which is a power of two. We provide a dictionary structure that on a given set of $n \leq N$ keys uses memory space of size $O(N)$, can be constructed in time $O(n \log \log N + N)$, and performs lookups in $O(1)$ time. Hence, for universes of size $O(n^2 / \log^2 n)$ such a structure immediately satisfies the desired performance. In case $\Omega(n^2 / \log^2 n) \leq |U| \leq n^{O(1)}$ we may use a sequence of dictionary structures of the same type. The first dictionary is built over the projection of S on the first $2 \log n - 2 \log \log n$ bits. If the size of the projected set is n_1, then each projected value can be assigned a unique identifier in the set $[n_1]$. During lookups these identifiers can be retrieved using the dictionary. The second dictionary is built over elements that are formed by concatenating the associated identifier and the projection of original key on the next $2 \log n - \log n_1 - 2 \log \log n$ bits. This process is continued until all bits are exhausted. Since $|U| = n^{O(1)}$ there is a constant number of dictionaries in the sequence.

A possible practical optimization is to handle small subsets directly. Namely, if an identifier value corresponds to a "small" number of elements of S, a specialized structure can store those elements, and they skip the rest of the general procedure. There are structures that can very efficiently handle sets of size $\log^{O(1)} N$ (see Section 3.2.8).

3.2.3 Central part

Here we give a high-level description of the construction. Explanations of subprocedures and second-level structures follow in later subsections. Let $\Psi = \log N$ and $\Phi = \lceil \log N - 2 \log \log N \rceil$. Suppose that $\phi : U \to \{0,1\}^\Phi$ and $\psi : U \to \{0,1\}^\Psi$ are functions such that the combined function (ϕ, ψ) is 1-1 on U. An easy choice is to take ϕ to be the projection on the Φ highest order bits, and ψ to be the projection on the Ψ lowest order bits of binary representations of keys. We have that $\Phi + \log \Phi + \log \Psi \leq \Psi + 1$ and $\log n \leq \Psi$.

The main hash function is of type

$$h(x) = \psi(x) \oplus a_{\phi(x)} \ ,$$

where (a_i) is an array of Ψ-bit elements, with $i \in \{0,1\}^\Phi$. Our aim is to set the values of the array elements in a way that makes the function h have no more than $3\Phi^2 n$ collisions on a given set $S \subset U$. It will become clear that this is always possible. After the function h is fixed, buckets of elements colliding under h need to be resolved. This is much easier than the original problem, since the average size of buckets is small. If the size of a bucket is less than $\Phi^3 \Psi$ then a structure specialized for small sets will handle it. The total number of elements in the remaining ("large") buckets is $O(\frac{n}{\Phi\Psi})$. This can be seen by analyzing function $\sum_i b_i$ under constraint $\sum_i \binom{b_i}{2} \leq 3\Phi^2 n$ and with variable domains $[\Phi^3 \Psi, \infty)$. Let S' be the subset of S comprising the elements that fall in the "large" buckets. Constructing an efficient dictionary over S' will be an easier task, because we can afford to spend $O(|S'|\Phi\Psi + 2^\Psi)$ construction time on it; we cover this in Section 3.2.9. No additional new techniques are required to design these second-level structures.

We will now give an overview of the algorithm for selecting values for the elements of the array a. The array a is initially set to all-zeros. Values of array elements will be decided in stages, with each stage being responsible for a separate set of bit positions. In our numbering of bit positions, position 0 refers to the most-significant bit position. Let $i_* = \lfloor \log \Psi - \log \log \Phi - 1 \rfloor$. There will be a total of $2i_* + 2 = O(\log \Psi)$ stages. In the stages numbered $1, 2, \ldots, i_*$ the sizes of *active* sets of bit positions decrease roughly geometrically, while in the remaining $i_* + 2$ stages they have the same (small) size. Let $p_0 = 0$, $p_i = \lfloor (1 - 2^{-i})\Psi \rfloor - i \cdot \lfloor \log \Phi \rfloor$ for $0 < i \leq i_*$, and $p_i = p_{i-1} + \lfloor \log \Phi \rfloor$ for $i_* < i \leq 2i_* + 2$. In the ith stage bits at positions between p_{i-1} and $p_i - 1$ (inclusive) are decided on all elements of a.

The last $i_* + 2$ stages could be replaced with different and shorter sequences. Yet, in this presentation of the algorithm we keep the chosen setting because it is relatively simple and incurs a relatively small increase in the overall constant factor. Operation in all the stages is done by the same procedure, parameterized by values p_{i-1} and p_i. We introduce symbols η_i denoting $2^{p_i - p_{i-1} + \lfloor \log \Phi \rfloor}$.

After the ith stage of the algorithm, the projection of $h(x)$ on the high-order p_i bits is known. In other words, for any $x \in U$ the value of $h(x)$ div $2^{\Psi - p_i}$ is fixed after the ith stage. To describe operation of the algorithm in stage i, we will define sets $T(v, j, k)$, $v \in \{0,1\}^{p_{i-1}}$, $0 \leq j \leq \Phi$, $0 \leq k < 2^{\Phi - j}$ (whenever we talk about sets $T(v, j, k)$ the stage number i is assumed to be fixed). Sets $T(v, j, k)$ are defined recursively as follows:

- For any $v \in \{0,1\}^{p_{i-1}}$, $T(v, \Phi, 0) = \{x \in S \mid h(x) \text{ div } 2^{\Psi - p_{i-1}} = v\}$.

- For $j < \Phi$, if $|T(v, j+1, k \text{ div } 2)| < \eta_i$ then $T(v, j, k) = \emptyset$.

- For $j < \Phi$, if $|T(v, j+1, k \text{ div } 2)| \geq \eta_i$ then

$$T(v, j, k) = \{x \in T(v, \Phi, 0) \mid k2^j \leq \phi(x) < (k+1)2^j\} \ .$$

Only non-empty sets $T(v, j, k)$ are of interest to us. For any fixed v, subset relation on the family of non-empty sets $T(v, j, k)$ can be described by a binary tree, with nodes labeled by pairs (j, k). Sets $T(v, j, k)$ that correspond to leaves of that tree are those that satisfy $j = 0$ or $T(v, j-1, 2k) \cup T(v, j-1, 2k+1) = \emptyset$. Let $\{S_{vl}\}_{v,l}$ be the collection of all such "leaf" sets, over $v \in \{0,1\}^{p_{i-1}}$. The collection $\{S_{vl}\}$ is a partition of the set S.

No matter how the elements of the array a are modified in current and later stages, that is on bit positions from p_{i-1} to $\Psi - 1$, the number of collisions that h may create is bounded by $\sum_v \sum_{l_1 < l_2} |S_{vl_1}| \cdot |S_{vl_2}|$ plus a bound on the total number of collisions within the sets S_{vl}. If a set S_{vl} has size greater than η_i then it has to be one of the sets $T(v, 0, k)$. However, the set $\{x \in S \mid \phi(x) = k\} \supset T(v, 0, k)$ is always mapped injectively by h. This follows from the definition of the function h, the fact that (ϕ, ψ) is 1-1 on U, and the properties of xor operation. Therefore collisions may happen only within the sets S_{vl} such that $|S_{vl}| < \eta_i$. An upper bound on the total number of collisions that may happen within the sets S_{vl} is $\frac{1}{2} n \eta_i$, which can easily be seen by analyzing function $\frac{1}{2} \sum_{j=1}^n b_j^2$ under constraint $\sum_j b_j = n$ and over domain $[0, \eta_i]^n$. The goal of processing in stage i

51

is to modify the values in the array a so that the number of collisions of h on S does not exceed

$$\eta_i \frac{n}{2} + \frac{1}{2^{p_i - p_{i-1}}} \sum_v \sum_{l_1 < l_2} |S_{vl_1}| \cdot |S_{vl_2}| \ , \tag{3.1}$$

when the stage ends. By solving appropriate recurrences, the following technical lemma can be shown. The proof is in Section 3.2.4.

Lemma 3.2.1. *If modifications to the array a by the selection algorithm make the number of collisions of h on S not exceed (3.1) at the end of stage i, for each i, then the final number of collisions will be less than $3\Phi^2 n$.*

The term $\sum_v \sum_{l_1 < l_2} |S_{vl_1}| \cdot |S_{vl_2}|$ from (3.1) can be re-expressed in an algorithmically more useful form. Each set $T(v, j, k)$ is the union of some sets S_{vl}. Thus, we may write $|T(v, j, k)| = \sum |S_{vl}|$, where the sum is over all l such that $S_{vl} \subset T(v, j, k)$. The product $|S_{vl_1}| \cdot |S_{vl_2}|$, for some l_1, l_2, will be a term in the expanded expression for a product of type $|T(v, j, 2k)| \cdot |T(v, j, 2k+1)|$. Actually it will appear as a component of exactly one such product; in the mentioned binary tree, the node with label $(j+1, k)$ has to be the lowest common ancestor of the nodes that correspond to the sets S_{vl_1} and S_{vl_2}. As a result, it holds that

$$\sum_v \sum_{l_1 < l_2} |S_{vl_1}| \cdot |S_{vl_2}| = \sum_v \sum_{j=0}^{\Phi-1} \sum_{k=0}^{2^{\Phi-j-1}-1} |T(v, j, 2k)| \cdot |T(v, j, 2k+1)| \ .$$

After we specified the goal of processing in every stage, we proceed to giving a high-level description of the sequence of operations done at each stage. We introduce multiset variables $X(v, j, k)$, and we implicitly initialize all of them to $\emptyset$. In the outermost loop of the procedure, j takes values from 0 to $\Phi - 1$. We describe principal operations performed for a fixed j. First, for all leaf sets $T(v, j, k)$, i.e. those that equal one of the sets S_{vl}, we make the assignment

$$X(v, j, k) = \{(h(x) \text{ div } 2^{\Psi - p_i}) \text{ mod } 2^{p_i - p_{i-1}} \mid x \in T(v, j, k)\} \ ,$$

where values $h(x)$ are taken to be determined by the current state of the array a. We effectively calculated the projections of the current values $h(x)$, $x \in T(v, j, k)$, on the bits at positions p_{i-1} through $p_i - 1$. The multisets can be stored as sets of element-multiplicity pairs. For each k, $0 \le k \le 2^{\Phi-j-1} - 1$, the algorithm will find a value $\delta \in \{0, 1\}^{p_i - p_{i-1}}$ such that

$$\sum_v \text{coll}(X(v, j, 2k), X(v, j, 2k+1) \oplus \delta) \le \frac{1}{2^{p_i - p_{i-1}}} \sum_v |T(v, j, 2k)| \cdot |T(v, j, 2k+1)|$$

and then make assignments $X(v, j+1, k) = X(v, j, 2k) \cup (X(v, j, 2k+1) \oplus \delta)$, where the union is in the multiset sense. The subprocedure that finds a suitable value δ is described in Section 3.2.6. The elements of the array a are modified so that $a_l = a_l \oplus 0^{p_{i-1}} \delta \, 0^{\Psi - p_i}$, for $(2k+1)2^j \le l < (2k+2)2^j$. At the end of the current iteration of the loop over j, the equality

$$X(v, j+1, k) = \{(h(x) \text{ div } 2^{\Psi - p_i}) \text{ mod } 2^{p_i - p_{i-1}} \mid x \in T(v, j+1, k)\}$$

holds for every non-leaf set $T(v, j + 1, k)$.

It is not hard to formally verify that a procedure conforming with this high-level description meets the specified goal of reducing the number of collisions of the function h on the set S. Performance analysis is completed in Section 3.2.7. We mention here that the following fact is used.

Lemma 3.2.2. *There can be at most $4n\frac{\Phi+1}{\eta_i}$ non-empty multisets $X(v, j, k)$.*

3.2.4 The proof of Lemma 3.2.1

An upper bound on the number of possible collisions after completion of stage i is given by the following recurrence for $i \leq i_*$:

$$c_0 = \frac{n^2}{2} , \quad c_i = 2^{p_i - p_{i-1}} 2^{\lfloor \log \Phi \rfloor} \frac{n}{2} + \frac{1}{2^{p_i - p_{i-1}}} c_{i-1} .$$

The solution to the recurrence is

$$c_i = 2^{\lfloor \log \Phi \rfloor} \frac{n}{2} \sum_{j=1}^{i} \frac{2^{p_j - p_{j-1}}}{2^{p_i - p_j}} + \frac{1}{2^{p_i}} \frac{n^2}{2} .$$

According to the definition of the sequence (p_i), the difference $p_i - p_j$, for $j < i \leq i_*$, satisfies the following inequality.

$$|(p_i - p_j) - (2^{-j} - 2^{-i})\Psi + (i - j)\lfloor \log \Phi \rfloor| \leq 1 . \tag{3.2}$$

Using (3.2) we may bound $\frac{2^{p_j - p_{j-1}}}{2^{p_i - p_j}}$ by $4 \cdot 2^{2^{-i}\Psi + (i-j-1)\lfloor \log \Phi \rfloor}$. As a result,

$$c_i \leq 4 \cdot 2^{2^{-i}\Psi} \frac{n}{2} \sum_{j=0}^{i-1} 2^{j\lfloor \log \Phi \rfloor} + \frac{2^{i\lfloor \log \Phi \rfloor}}{2^{(1-2^{-i})\Psi}} n^2 .$$

A simple and relatively tight upper bound on c_{i_*} is $2^{2^{-i_*}\Psi} 2^{i_*\lfloor \log \Phi \rfloor}(n + \frac{n^2}{2^\Psi})$. We use this value as the starting point in the recurrence that corresponds to the remaining stages:

$$\bar{c}_0 = 2^{2^{-i_*}\Psi} 2^{i_*\lfloor \log \Phi \rfloor} \left(n + \frac{n^2}{2^\Psi}\right) , \quad \bar{c}_i = 2^{2\lfloor \log \Phi \rfloor} \frac{n}{2} + \frac{1}{2^{\lfloor \log \Phi \rfloor}} \bar{c}_{i-1} .$$

The solution to this recurrence is

$$\bar{c}_i = 2^{2\lfloor \log \Phi \rfloor} \frac{n}{2} \sum_{j=0}^{i-1} \frac{1}{2^{j\lfloor \log \Phi \rfloor}} + \frac{1}{2^{i\lfloor \log \Phi \rfloor}} \bar{c}_0 .$$

After the whole procedure of selecting values for the elements of a is finished, the number of collisions of h on S is no more than $\bar{c}_{i_*+2} < n\Phi^2(2 + 2^{-\Psi}n)$. From $\Psi \geq \log n$ it follows that $\bar{c}_{i_*+2} < 3\Phi^2 n$, as required at the beginning.

3.2.5   Determining and arranging sets S_{vl}

We need a procedure that efficiently determines which of the sets $T(v, j, k)$ are leaf sets, that is, those that equal one of the sets S_{vl}. This procedure needs to be executed at the beginning of every stage of the algorithm. The output should be a list of triples of form (v, j, k). Naturally, the first step of the procedure is to sort the elements of S according to values of the function $(h(x) \text{ div } 2^{\Psi - p_{i-1}}, \; \phi(x))$. One way of determining leaf sets $T(v, j, k)$ is to build *path-compressed* tries. A trie would be built for each value of $h(x) \text{ div } 2^{\Psi - p_{i-1}}$ separately. By computing and storing weights of subtrees at each internal node, it is easy to determine the leaf sets recursively by the definition. This method would be efficient in the word RAM model, but we would have problems making it worst-case efficient in the external memory models.

We constructed a different and somewhat simpler algorithm for this problem, which is efficient both in terms of RAM time and I/O cost. The pseudo-code of the procedure for determining leaf sets $T(v, j, k)$ is listed in Algorithm 1. Apart from the initial sorting, it runs in $O(n)$ time making $O(n/B)$ I/Os. In terms of external memory computation, effectively two scans of the array are made; it is possible to do only a single scan with a small complication. The algorithm is not completely trivial, but it is not a problem to prove its correctness. We leave the tedious details out. Performance in the external memory setting is obvious. The only non-obvious points in time analysis for the RAM model are the two while loops that increase/decrease values of j and k. We charge each iteration of these loops to the latest output leaf set that contained at least $\frac{1}{2}\eta_i$ elements. The crucial observation is that a total of $O(\Phi)$ iterations may be charged to a set, possibly split between different executions of the loops. Since $\eta_i > \Phi$, for any i, we are done.

So far we have the labels of the leaf sets $T(v, j, k)$ in form of triples (v, j, k), and the set S is stored so that the elements of each set $T(v, j, k)$ appear at consecutive cells. As a preparation for subsequent computations, the sets are separated into Φ groups according to the value of the index j. Consider the group $\bar{j}$, for some $0 \leq \bar{j} < \Phi$. The sets in group $\bar{j}$ do not participate in computations during the first $\bar{j}$ iterations of the outer loop in stage i, i.e. for $0 \leq j < \bar{j}$. When j reaches value $\bar{j}$ it is time to include those sets in processing, which requires the corresponding multisets $X(v, \bar{j}, k)$ to be computed (remember that we are dealing only with leaf sets at this point). Remark that the description of the function h has changed since the start of stage i, and therefore this computation cannot be performed at an earlier time, before the iteration number $\bar{j}$. In the external memory setting the computation of the multisets $X(v, \bar{j}, k)$ involves sorting of all elements in $\cup_{v,k} T(v, \bar{j}, k)$, where the union only includes the leaf sets, according to values of the function ϕ, in order to compute the values of the function h on those elements. The function values are then computed by making linear scans over the obtained sorted sequence and the array a. Another sorting operation is used to group elements of each set $X(v, \bar{j}, k)$ together, with the sets laid out sorted according to the values of indices k and v, respectively (having $\phi(x)$ as the first attribute in a sorting key is not the same as having k-value as the first attribute when the key is composite).

$(x_q) \leftarrow$ sequence of elements of S sorted according to values of the function $(h(x) \operatorname{div} 2^{\Psi - p_{i-1}}, \phi(x))$;

$j \leftarrow \Phi$;
$k \leftarrow 0$;
$v \leftarrow h(x_1) \operatorname{div} 2^{\Psi - p_{i-1}}$;
$count \leftarrow 1$;
$q_{start} \leftarrow 1$;

for $q \leftarrow 2$ **to** n **do**
    if $v \neq h(x_q) \operatorname{div} 2^{\Psi - p_{i-1}}$ **then**
        if $count > 0$ **then**
            Append triple (v, j, k) to the output list;
        $j \leftarrow \Phi$;
        $k \leftarrow 0$;
        $v \leftarrow h(x_q) \operatorname{div} 2^{\Psi - p_{i-1}}$;
        $count \leftarrow 1$;
        $q_{start} \leftarrow q$;
    else if $\phi(x_q) \geq (k+1)2^j$ **then**
        if $count > 0$ **then**
            Append triple (v, j, k) to the output list;
        while $(k+2)2^j < \phi(x_q)$ **do**
            $j \leftarrow j + 1$;
            $k \leftarrow k \operatorname{div} 2$;
        end
        $k \leftarrow k + 1$;
        $count \leftarrow 1$;
        $q_{start} \leftarrow q$;
    else
        $count \leftarrow count + 1$;
        if $count \geq \eta_i$ **then**
            while $j > 0 \ \wedge \ (k+1)2^j > \phi(x_q)$ **do**
                $j \leftarrow j - 1$;
                $k \leftarrow 2k$;
            end
            if $(k+1)2^j \leq \phi(x_q)$ **then**
                if $\phi(q_{start}) < (k+1)2^j$ **then**
                    Append triple (v, j, k) to the output list;
                    while $\phi(q_{start}) < (k+1)2^j$ **do**
                          $q_{start} \leftarrow q_{start} + 1$;
                          $count \leftarrow count - 1$;
                    end
                $k \leftarrow k + 1$;
end

<div align="center">

Algorithm 1: Finding leaf sets $T(v, j, k)$

</div>

3.2.6   Finding suitable δ values and merging the multisets

We do not store the multisets $X(v, j, k)$ in the simplest form. To facilitate bit-by-bit selection of δ values, we store weighted full binary tree representations of the multisets $X(v, j, k)$. Namely, for each multiset $X(v, j, k)$ we store an array of length $s_i = 2^{p_i - p_{i-1} + 1} - 1$. Let $(w_q)_{q=1}^{s_i}$ denote such an array. Then for $2^{p_i - p_{i-1}} \leq q \leq s_i$, w_q has the value of the multiplicity of element $q - 2^{p_i - p_{i-1}}$ in the multiset. The multiplicity of an element that does not belong to the multiset is zero. For $1 \leq q < 2^{p_i - p_{i-1}}$, w_q has the value of $w_{2q} + w_{2q+1}$.

Applying the operator $\oplus$ with argument δ on $X(v, j, k)$ has the following effect on its weighted binary tree representation: if the bit at position l of δ has value 1 then for every node of level l its left and right subtree are interchanged (the root is at level 0). Each level in the weighted tree for the set $X(v, j, k) \oplus \delta$ is a permutation of the same level in the tree for $X(v, j, k)$. For level l, the first l bits of δ determine the permutation.

For leaf sets $T(v, j, k)$ the weighted tree representations of the corresponding multisets $X(v, j, k)$ are constructed after computation and grouping of the sets $X(v, j, k)$, mentioned in Section 3.2.5. It will be convenient to have the arrays of weights split into segments corresponding to levels of the trees, and to group together segments for each level. There are $p_i - p_{i-1}$ groups, and within each group segments are arranged sorted according to the values of indices k and v of their sets $X(v, j, k)$. All this preprocessing of data structures is done for the multisets $X(v, \bar{j}, k)$ that correspond to the leaf sets $T(v, \bar{j}, k)$ in the $(\bar{j} + 1)$-st iteration of the outer loop in stage i. A result of the first $\bar{j}$ iterations of the loop are the same kind of data structures for the multisets $X(v, \bar{j}, k)$ that correspond to non-leaf sets $T(v, \bar{j}, k)$, also laid out in the same way. As the last preparation step in the $(\bar{j} + 1)$-st iteration, before doing the main operations, those two sets of data structures are merged into one. Since in both parts all pieces are already arranged sorted according to tree-level number, and then indices k and v, no sorting operation is involved in this step.

Recall of our principal task at this place — for fixed j and k we need to select δ such that the following relation is satisfied.

$$\sum_v \mathrm{coll}(X(v, j, 2k),\ X(v, j, 2k + 1) \oplus \delta) \leq$$

$$\frac{1}{2^{p_i - p_{i-1}}} \sum_v |T(v, j, 2k)| \cdot |T(v, j, 2k + 1)| \tag{3.3}$$

Initially δ is set to 0, and in increasing order of bit positions it is decided whether to set the bit to 1 or not. Since in each problem the values of j and k are fixed, we may use (w_{vq}) to denote the weight arrays of the multisets $X(v, j, 2k)$, and (w_{vq}^δ) to denote the weight arrays of the multisets $X(v, j, 2k + 1) \oplus \delta$. Define the function

$$\mu(l, \delta) = \sum_v \sum_{q=2^l}^{2^{l+1} - 1} w_{vq} \cdot w_{vq}^\delta \ .$$

For any δ, the value of $\mu(p_i - p_{i-1}, \delta)$ is an upper bound on

$$\sum_v \text{coll}(X(v,j,2k), \ X(v,j,2k+1) \oplus \delta) \ .$$

Therefore it is sufficient to set the bits of δ so that $\mu(l+1, \delta) \leq \frac{1}{2}\mu(l, \delta)$, for every l (only the first l bits of δ influence $\mu(l, \delta)$). Let δ_l be the value obtained by taking the first l bits of δ and setting the remaining bits to zeros. By expanding w_{vq} and w_{vq}^{δ} into $w_{v\,2q} + w_{v\,2q+1}$ and $w_{v\,2q}^{\delta_l} + w_{v\,2q+1}^{\delta_l}$, we get that

$$\mu(l, \delta) \ = \ \sum_v \sum_{q=2^{l+1}}^{2^{l+2}-1} (w_{v\,q} \cdot w_{v\,q}^{\delta_l} + w_{v\,q+1} \cdot w_{v\,q+1}^{\delta_l}) +$$

$$+ \sum_v \sum_{q=2^{l+1}}^{2^{l+2}-1} (w_{v\,q} \cdot w_{v\,q+1}^{\delta_l} + w_{v\,q+1} \cdot w_{v\,q}^{\delta_l}) \ .$$

We expressed $\mu(l, \delta)$ as the sum of two terms such that: if the bit at position l of δ is set to 0 then $\mu(l+1, \delta)$ becomes equal to the first term, and otherwise if it is set to 1 then $\mu(l+1, \delta)$ becomes equal to the second term. The choice is made based on which value is smaller.

When deciding on the value of bit at position l, the procedure needs to permute the sequence $(w_{vq}^0)_{q=2^{l+1}}^{2^{l+2}-1}$ according to δ_l to get $(w_{vq}^{\delta_l})_{q=2^{l+1}}^{2^{l+2}-1}$, for each v. In the word RAM model this is easy to do in $O(2^{l+1})$ time, while in external memory sorting is required. Joining the weighted binary tree representations of multisets $X(v,j,2k)$ and $X(v,j,2k+1) \oplus \delta$ into the same type of representation for the multiset $X(v,j+1,k)$ is straightforward. It can even be done on-the-fly during the selection of bits of δ. If the data structures for $X(v,j+1,k)$ are stored at the same places were the structures for $X(v,j,2k)$ resided, then the memory that stored the structures for $X(v,j,2k+1)$ becomes available. The space should be compacted to contain no unused holes. The compaction can be done either at the end of processing in the iteration number j, or on-the fly during the other computations.

Total time spent in stage i on all operations covered in this subsection can be expressed as $O(2^{p_i - p_{i-1}}) = O(\frac{\eta_i}{\Phi})$ per every non-empty multiset $X(v,j,k)$. When counting the number of I/Os in the external memory version, note that sorting operations in this part of the algorithm are conducted as procedures of multi-sorting of groups of elements of equal sizes. All other operations involve only sequential scans. The total number of I/Os made in stage i for the operations covered in this subsection can be expressed as $O(\text{Sort}(2^{p_i - p_{i-1}})) = O(\frac{1}{\Phi}\text{Sort}(\eta_i))$ per every non-empty set $X(v,j,k)$.

3.2.7 Completing the analysis of the main algorithm

We analyze the total cost of one stage of the algorithm for selecting values for the elements of the array a. Aggregate complexity of the procedures specified in Section 3.2.5 is proportional to the complexity of sorting n integers of size $O(\log N)$ bits. In the external memory setting the limited size of numbers does

not help us (at least it is not known to), and the required complexity is simply $O(\text{Sort}(n))$ I/Os. In the version for the word RAM model we choose to use radix sort when $n \geq N/\log^2 N$. Then, using a space of size $O(n)$ sorting takes time $O(n)$. When $n < N/\log^2 N$ we use a comparison-based sorting sorting procedure. In any case, a time bound for this part is $O(n + N/\log N)$.

In Section 3.2.6 it was said that the total time spent in stage i on all the operations from that part can be expressed as $O(\frac{\eta_i}{\Phi})$ per every non-empty set $X(v, j, k)$. Lemma 3.2.2 implies that the operations described in Section 3.2.6 run in total time $O(n)$ in any stage. In the cache-oblivious model the cost is $O(\frac{n}{B} \log_{M/B} \eta_i) = O(\text{Sort}(n))$ I/Os.

Proof of Lemma 3.2.2. Consider any fixed v and the set $T(v, \Phi, 0)$ with its associated binary tree that captures the subset relation on non-empty sets $T(v, j, k)$. In the degenerate case $|T(v, \Phi, 0)| < \eta_i$ the tree is just the root node, and $X(v, j, k) = \emptyset$ for any j, k ($X(v, \Phi, 0)$ is also empty since no assignment involving that variable is made in the course of the algorithm; the outermost loop goes until $j = \Phi - 1$). Suppose that $|T(v, \Phi, 0)| \geq \eta_i$. Since the collection of leaf sets $T(v, j, k)$ is a partition of $T(v, \Phi, 0)$, among the leaf sets there can be at most $2|T(v, \Phi, 0)|/\eta_i$ sets having size at least $\eta_i/2$. Any non-leaf set has a descendant leaf set of size at least $\eta_i/2$, which is straightforwardly derived from the definition of sets $T(v, j, k)$. We "assign" each non-leaf node to one of its descendant leaf node sets of size at least $\eta_i/2$ (multiple nodes can be assigned to a leaf node). Further, we assign each leaf node (j, k) such that $|T(v, j, k)| < \eta_i/2$ to one leaf descendant (j_1, k_1) of its parent for which $|T(v, j_1, k_1)| \geq \eta_i/2$. Each leaf node (j, k) such that $|T(v, j, k)| \geq \eta_i/2$ can have at most $2(\Phi - j)$ nodes assigned to it — at most two per every level on the path to the root. Consequently, there can be at most $(2\Phi + 1)\frac{2}{\eta_i}|T(v, \Phi, 0)|$ non-empty sets $X(v, j, k)$, for the chosen fixed v. Summing over all v finishes the proof. $\square$

The only remaining thing is to take into account the operations of modifying the elements of a, that is, the operations of type

$$a_l = a_l \ \oplus \ 0^{p_{i-1}} \delta \, 0^{\Psi - p_i} \ .$$

Every element gets modified Φ times in every stage of the algorithm; therefore $O(2^\Phi \Phi)$ time is spent on this process in one stage. Because of the inequality $\Phi + \log \Phi + \log \Psi \leq \Psi + 1$, a time bound is $O(2^\Psi/\Psi) = O(N/\log N)$. The elements of (a_l) are modified in order of increasing index l. Hence, I/O cost is $O(\frac{N}{B \log N})$.

Even if the sequence (p_i) was changed, possibly resulting in a higher number of stages, there could be at most $\Psi = \log N$ stages. Hence, the bounds which were expressed only in terms of N yield a combined bound of $O(N)$ over the entire algorithm. With the chosen parameters there are $O(\log \Psi) = O(\log \log N)$ stages, implying a complete time bound of $O(n \log \log N + N)$. In the cache-oblivious model the algorithm makes $O(\text{Sort}(n) \log \log N + N/B)$ I/Os.

3.2.8 Subsets of size $\log^{O(1)} N$

In the word RAM model, a generalization of *fusion trees* of Fredman and Willard [FW93] yields a linear-space static dictionary that on a set of m keys has a lookup time of $O(1 + \frac{\log m}{\log w})$ and it can be constructed in $O(m)$ time on sorted input (this was explicitly stated by Hagerup [Hag98b]). Plugging in $m = \log^{O(1)} N$ and $w = \Omega(\log N)$ shows that lookup time would be $O(1)$ in our case. Sorted sequence can be obtained through radix-sorting elements of type (bucket-id, x) from all the small buckets.

Alternatively, we may use a combination of our deterministic signatures method (Chapter 2) and packed B-trees (e.g. see [And95]). This combination is a bit simpler than fusion trees, and it can also be used in external memory. A signature function that maps keys to $O(\log \log N)$ bit values can be found in linear time on sorted input. The signatures of size $O(\log \log N)$ bits are stored in a packed B-tree. Since the word size is $\Omega(\log N)$ bits, the tree will have a constant depth.

3.2.9 The secondary structure for large buckets

On a RAM we could resort to using the structure for polynomial-size universes from [HMP01]. Because there are $o(n/\log n)$ elements at this part, the added construction cost is $o(n)$. However, in external memory this structure is inefficient to construct.

We will use a variation of our algorithm to cut the number of collisions on these elements down to $o(n/\log n)$. Then within the buckets created by this second-level structure we use hash functions with quadratic range; quadratic space can be afforded in each bucket because the sum of collisions is small. This approach is the same as in the FKS scheme [FKS84]. To find injective hash functions deterministically we may use the algorithm from [Ram96] or the algorithm from Chapter 4. The former one would run in time $O(m^2 \log n)$ on a bucket of size m, while the latter one would take time $O(m^2 \log m)$. In any case the combined time is $o(n)$. These algorithms can easily be made efficient in the cache-oblivious model.

It remains to describe changes to some parameters from Section 3.2, which will allow a higher drop in the number of collisions. The parameters that correspond to sequences (p_i) and (η_i) will be set differently. Now there will be Ψ stages. We simply set $\hat{p}_i = i$ and $\hat{\eta}_i = 1$, for $0 \leq i \leq \Psi$. The algorithm is obtained by substituting $\hat{p}_i$ and $\hat{\eta}_i$ for p_i and η_i, respectively. Except from some parts of the analysis, everything else stays the same. The recurrence for the number of possible collisions after completion of stage i is now simply: $\hat{c}_0 = O((\frac{n}{\Phi \Psi})^2)$, $\hat{c}_i = \frac{1}{2}\hat{c}_{i-1}$. The final number of collisions is $O(n/(\Phi \Psi)^2) = O(n/\log^2 n)$. Performance bounds for procedures described in Section 3.2.6 grew by a factor of $O(\Phi \Psi)$, but this was amortized by having the input set of smaller size.

Remark that only a small fraction of the elements goes through the additional complications from this part.

3.3 Larger Universes

3.3.1 Background on signature functions

The basic type of functions used in Chapter 2 is $f(x, s, a) = x$ div $2^s + a \cdot (x \bmod 2^s)$, where a is a parameter chosen from $\{1, 2, \ldots, n^c - 1\}$, $c \geq 2$. The parameter s has a value dependent only on the domain of x, for example $s = \lfloor \frac{1}{2} \log |U| \rfloor$. The integer division and modulo functions were chosen as they are perhaps the simplest of all pairs of functions (ϕ, ψ) such that (ϕ, ψ) is 1-1 on U, and so that both functions map to a (significantly) smaller universe. In a more general form, we write $f(x, a) = \phi(x) + a \cdot \psi(x)$. Suppose that K is the number of keys that can be packed in a machine word. With $c = 3.42$, on any given set of n keys, a value for the parameter a that makes the function f injective on the set can be found in time $O(n(\log n)^2 \frac{\log K}{K} + (\log n)^3)$. The basic function can be combined in different ways to achieve a larger reduction of universe. The ultimate goal is to have a function that maps original keys to signatures of size $O(\log n)$ bits. Many concrete variants of this approach can be imagined, yet we need only two.

Let $x[0]x[1] \ldots x[q-1]$ be a string representation of key x over some alphabet. One possibility is to apply f to all characters and concatenate the resulting values, viewed as binary strings. We may use the same multiplier parameter for all characters, and thus the length-reduced value for key x after one level of reduction has a form of $f(x[0], a)f(x[1], a) \ldots f(x[q-1], a)$. The process is repeated with different multipliers and possibly different alphabets at subsequent levels of reduction. We refer to this way of combining function f as the *parallel reduction*. In the second version, which we call *suffix reduction*, only the last characters get reduced at a single reduction level. Although the structure of reduction sequences is different for those two variants, as well as the processes of parameter selection, the final functions for those two compositions can have similar *dot product* forms. A precondition for this is to suitably set intermediate alphabet sizes in the parallel reduction. For example, after two levels of parallel reduction we want the function to have a form of $x \mapsto f(f(x[0], a)f(x[1], a), a') \ldots f(f(x[q-2], a)f(x[q-1], a), a')$. Having the final functions in dot product form means that they can be evaluated rather efficiently on a word RAM. For more information about these methods see Chapter 2.

A final injective function is composed of functions generated by the method of parallel reduction and the method of suffix reduction. It can be evaluated in constant time on a word RAM. When $\log^{3+\epsilon} n < w < 2^{\frac{n}{\log n}}$ the construction algorithm runs in linear time on sorted input. We may use fusion trees to cover the extreme case $w \geq 2^{\frac{n}{\log n}}$ efficiently. In this chapter we show an improved complexity of dictionary construction in the case that $w \leq \log^{O(1)} n$.

3.3.2 Speed-up of the suffix reduction

The briefly outlined method of making deterministic signatures produces perfect hash functions with ranges of polynomial size. The functions have rather succinct descriptions, and they might have an application outside of dictionary

structures. Here we will give up the injectiveness requirement, and replace it with considerably weaker properties. These weaker functions will be useful only when combined with additional data structures, foremost a dictionary structure for universes of polynomial size. The variant of the suffix reduction method that we introduce is particularly efficient in the external memory models. Yet it also produces some useful results in the word RAM model.

Let $x[i]_\sigma$ denote the ith character of key x viewed as a string over the alphabet $[2^\sigma]$. It is assumed that $\sigma = \Omega(\log n)$. In the original version of suffix reduction we want to find multipliers $a_0, a_1, \ldots, a_{q-2}$ such that the function

$$a_0 \cdot x[0]_\sigma + (\ldots + (a_{q-3} \cdot x[q-3]_\sigma + (a_{q-2} \cdot x[q-2]_\sigma + x[q-1]_\sigma)) \ldots) \quad (3.4)$$

is injective on S. The multiplier selection algorithm is applied $q-1$ times, as suggested by the expression in (3.4). In the new version, the multiplier selection procedure is again called $q-1$ times, but each time with an input set of size $O(n/(\log n)^2)$. There will be no limit on the number of collisions that the final function may cause. Yet the function will have some properties that will allow the initial searching problem to be reduced either to a problem over a universe of size $O(\sigma)$ bits, or to a problem over a set of size $O(\log^2 n)$. The high level idea is to look for clusters of elements that already piled up and will hash to equal values by the final function, and to prevent further collisions between already formed clusters. Sorting operations (over shorter keys) will dominate the running times. Suppose that values for the parameters $a_l, a_{l+1}, \ldots, a_{q-2}$ were selected. If two keys x and y share the prefix of length l and the function values on their suffixes of length $q-l$ collide, i.e. $a_l \cdot x[l]_\sigma + a_{l+1} \cdot x[l+1]_\sigma + \ldots + x[q-1]_\sigma = a_l \cdot y[l]_\sigma + a_{l+1} \cdot y[l+1]_\sigma + \ldots + y[q-1]_\sigma$, then x and y will certainly be mapped to the same value by the final function. On the other hand, if the length l prefixes of x and y differ, it does not matter what are the values of the partial function on their suffixes of length $q-l$, since the separation of their hash values will be decided at a later time. The construction algorithm will keep track of sufficiently large clusters of elements that are certain to collide given the already selected multipliers. Different clusters will be ensured to map to different values. However keys not yet belonging to any cluster are able to join existing clusters or form new ones. The time of joining a cluster for a given key, specified by a prefix length, is possible to determine quickly during lookups. Some pieces of information related to this joining point will enable us to substantially reduce the search space. To be precise, the reduced search space will consist of keys of length $O(\sigma)$ bits. If we set $\sigma = \Theta(\log n)$ then the method can be composed with the structure from Section 3.2. In some uses of the method, as we will see in Section 3.3.3, we need to set a higher value of σ; there the "pipelined" dictionary for keys of length $O(\sigma)$ bits is more complex.

The computationally dominant process in the construction algorithm will usually be sorting. The procedure performs $O(q)$ sorting operations over sets of $O(n)$ keys of length $O(\sigma)$ bits. In the external memory models this amounts to $O(\text{Sort}(n))$ I/Os (with the block size B expressed in terms of $(\log |U|)$-bit items, where U is the universe of keys that are input to the method). Combining this with the result from Section 3.2 produces the result stated in Theorem 3.1.2. In the word RAM model, the total sorting time is in general $O(nq \log \log n)$, based

on [Han04]. When $\sigma = \Theta(\log n)$ we may use radix sort and get a time bound of $O(nq)$, which explains Remark 3.1.5. When q is sufficiently large it makes sense to use a serial version of the parallel sorting algorithm from [AH92]; however, in our applications, suffix reduction is used with relatively small values of q.

We will mainly describe the process of constructing the functions, and through it provide understanding of the properties that the functions possess. The lookup procedure will become immediate once the construction is understood.

Suppose w.l.o.g that the sequence of the input keys is ordered as $x_1 < x_2 < \ldots < x_n$. As a preparation step, the procedure will assign some identifiers to prefixes of the keys. The identifiers need to be relatively small. Consider the set $\{x[0]_\sigma x[1]_\sigma \ldots x[l-1]_\sigma \mid x \in S\}$ of prefixes of length l of the keys. To each value in this set we want to assign a unique identifier from the set $[n]$. No relation is imposed between identifiers of prefixes of different lengths. For each element $x_k \in S$ we further want to have identifiers of its prefixes stored in a word-packed form, in the natural order. Assignment of prefix identifiers and storing of identifier sequences can be done simultaneously, doing an iteration through the sorted sequence of keys. For the first element we write the sequence $(0, 0, \ldots, 0)$. Suppose that the sequence of prefix identifiers for x_k is $(p_0, p_1, \ldots, p_{q-1})$. Let l be the length of the longest common prefix of x_k and x_{k+1}. Then for element x_{k+1} we write the sequence

$$(p_0, p_1, \ldots, p_l, \ p_{l+1} + 1, \ldots, p_{q-2} + 1, \ p_{q-1} + 1) \ .$$

The entire operation involving elements x_k and x_{k+1} can be executed in constant time using some standard techniques of computation using word-level parallelism. Let $p(k, l)$ denote the identifier value of prefix of x_k of length l.

We introduce a collection variable C and initialize it to $\emptyset$. During the construction procedure, every element of S will be assigned at most one element from C. Let $(c_i)_{i=1}^n$ be an array such that c_i holds a reference to the item from C assigned to x_i, if one exists; otherwise c_i has value -1.

For the first parameter in the sequence, which is a_{q-2}, we do not even need to run the selection algorithm; we simply put some value, e.g. $a_{q-2} = 1$. Then, the algorithm computes the set of triples

$$\{(p(k, q-2), \ a_{q-2} \cdot x_k[q-2]_\sigma + x_k[q-1]_\sigma, \ k)\}_{k=1}^n \ .$$

This set gets sorted in lexicographic order. The algorithm inspects subsets containing elements that match on the first two fields of the triples. Subsets of size less than $\log^2 n$ are ignored. Consider a subset of size $m \geq \log^2 n$. Let the value of the second field be y and the set of values at the third field be $\{k_1, k_2, \ldots, k_m\}$. The tuple $(k_1, \ y)$ is added to the collection C, and references to it are set on c_{k_i}, $1 \leq i \leq m$. No matter how the remaining parameters are selected, the elements $x_{k_1}, \ldots, x_{k_m}$ will be mapped to the same value by the final function.

Suppose that values for the parameters $a_{l+1}, a_{l+2}, \ldots, a_{q-2}$ were selected. We feed as input to the procedure that should select a value for a_l the set $\{(y, \ x_k[l]_\sigma) \mid (k, y) \in C\}$ (the input was represented as a set of pairs of values of functions ϕ and ψ). After the selection of a_l, the set of triples $\{(p(k, l), \ a_l \cdot x_k[l]_\sigma + a_{l+1} \cdot x_k[l+1]_\sigma + \ldots + x_k[q-1]_\sigma, \ k)\}_{k=1}^n$, is computed and then sorted

in lexicographic order. The algorithm inspects the subsets containing elements that match on the first two fields of the triples. Subsets of size less than $\log^2 n$ are ignored. Consider a subset of size $m \geq \log^2 n$. Let the value of the second field be y and the set of values at the third field be $\{k_1, k_2, \ldots, k_m\}$. If $c_{k_i} = -1$ for all $i \in \{1, \ldots, m\}$, then the tuple (k_1, y) is added to C, and references to it are set on c_{k_i}, $1 \leq i \leq m$. Otherwise, suppose that $c_{k_j} \neq -1$. According to our construction, if $c_{k_i} \neq -1$ for any $i \in \{1, \ldots, m\}$, then it must be $c_{k_i} = c_{k_j}$. The assignments $c_{k_i} = c_{k_j}$ are made for all $i \neq j$, and the second field of the associated tuple in C is changed to y.

Denote the final function by g. If $q = n^{O(1)}$ the range of g has a size of $O(\sigma)$ bits. The set $\{g(x) \mid x \in S\}$ is plugged in as the input to a dictionary for universe of size $O(\sigma)$ bits. For any i, $1 \leq i \leq n$, it holds that either $|\{x \in S \mid g(x) = g(x_i)\}| < \log^2 n$ or $c_i \neq -1$. The former case can be directly handled by a specialized structure. In the latter case, let (k, y) be the element of C referenced by c_i. The index k becomes the associated attribute of the hash value $g(x_i) = g(x_k)$.

For keys $x \in U$ whose hash value $g(x)$ falls in one of the buckets of size no less than $\log^2 n$, define function $\bar{g}$ by

$$\bar{g}(x) = (l, \ p(k,l), \ x[l], \ a_{l+1} \cdot x[l+1]_\sigma + \ldots + x[q-1]_\sigma) \ ,$$

where k is the index associated with value $g(x)$, and l is the length of the longest common prefix between x and x_k. It is not hard to prove that $|\{x \in S \mid \bar{g}(x) = y\}| < \log^2 n$, for any y. The elements $\bar{g}(x)$ are stored in another dictionary for universes of size $O(\sigma)$ bits, for keys $x \in S$ whose hash value $g(x)$ falls in one of the "large" buckets. With each stored value $\bar{g}(x)$ we associate a reference to a dictionary specialized for sets of size $\log^{O(1)} n$.

Examining the performance complexity of the presented method is easy. Supposing that the pipelined dictionary for keys of length $O(\sigma)$ bits has a lookup cost of $O(1)$, the entire cost of lookup is (a larger) constant. Excluding the construction complexity of the secondary dictionaries, sorting processes usually dominate the cost of the construction procedure (only when q is extremely large will the cost of the basic procedure that selects $a_0, \ldots, a_{q-2}$ outweigh the cost of sorting in the main procedure, and we never use this method for large q). The main procedure performs $O(q)$ sorting operations over sets of $O(n)$ keys of length $O(\sigma)$ bits.

It is interesting to observe that recursively applying the above reduction method a constant number of times leads to a dictionary with a construction time of $O(n \log^\epsilon n)$ on keys of length $\log^{O(1)} n$ bits, for any fixed $\epsilon > 0$; the lookup time is constant for a fixed ϵ, but grows quickly as ϵ decreases towards 0. A theoretically superior approach is outlined in Section 3.3.3.

3.3.3 Speed-up of the parallel reduction

This section covers the remaining case that $\omega(\log n \log \log n) < w < \log^{3+\epsilon} n$ to complete Theorem 3.1.3. The approach is conceptually very similar to the approach that led to the speed-up of the method of suffix reduction, but some

details are different and the computation is more involved. Because of the similarities we give only an outline; it should be enough for understanding, provided that Section 3.3.2 is understood (as well as some of the prior work on our signature functions, on which everything is based).

Consider partially reduced keys, after some number of levels of the parallel reduction. We call a prefix value heavy if it is shared by at least $(\log n)^2$ partially reduced keys. We ensure that the current level of reduction avoids any collisions between heavy prefixes. For this purpose, it is convenient to maintain the trie of the partially reduced set (see [Ruž07]). It is again possible to substantially reduce the search space for a given key by using information related to the key's point of joining a cluster of piled up elements.

In order to determine the level at which a key joined a cluster in constant time, we need to evaluate partially reduced values of the key for all levels in constant time. This is possible if $\Omega(K^2)$ copies of a key can fit in a single machine word, where K is the number of reduction levels. We explain this now. For simplicity, assume that $2K^2$ copies of a key can fit into one word; the assumption will easily generalize to $\Omega(K^2)$. The value of each partial evaluation is a concatenation of dot products (in a complete evaluation there is just one dot product). The constant operand is the same for all products (at one level), while the second operand is a segment of a key; different segment for different products. Because the constant operand is the same, all dot products can still be computed in constant time, using two multiplications and a few bitwise operations. The resulting value is not compacted — there are gaps with zero bits — but this is not a problem, they may stay this way. However, partial evaluations that go until different levels use different kinds of dot products, and the key needs to be prepared (split using bitwise AND) differently due to different alphabet sizes. Therefore, we write K equidistant copies of the key in one word. The other word contains constant operands of dot products for different levels, spaced at distance equal to two key sizes. There needs to be a lot of zero space between copies of the key in the first word to hold all K calculation results. Actually, for each copy there is one meaningful result which is kept; the others are by-products of the computation, and they are thrown away.

For $w = \log^{O(1)} n$ we have that $K = O(\log \log n)$. To provide a situation where $\Omega(K^2)$ copies of a key can fit into one word, we use two levels of the searching problem reduction from Section 3.3.2, using the setting $q = O(\log \log n)$. Hence the incurred construction cost from these two reduction steps is $O(n(\log \log n)^2)$.

The construction of this version of the parallel reduction function has a time cost proportional to K times the sorting time. For $w = \log^{O(1)} n$ we again get a bound of $O(n(\log \log n)^2)$. Since through the method of parallel reduction we map the keys to a range of size $O(\log n \log \frac{w}{\log n})$ bits, at the bottom end we again employ the suffix reduction, but this time paired with the dictionary for polynomial-size universes.

Chapter 4

Uniform Deterministic Dictionaries

Abstract

We present a new analysis of the well-known family of multiplicative hash functions, and improved deterministic algorithms for selecting "good" hash functions. The main motivation is realization of deterministic dictionaries with fast lookups and reasonably fast updates. The model of computation is the word RAM, and it is assumed that the machine word size matches the size of keys in bits. Many of the modern solutions to the dictionary problem are weakly non-uniform, i.e. they require a number of constants to be computed at "compile time" for stated time bounds to hold. In contrast, our dictionaries do not require any special constants or instructions, and running times are completely independent of the word (and key) length. Our family of dynamic dictionaries achieves a performance of the following type: lookups in time $O(t)$ and updates in amortized time $O(n^{1/t})$, for an appropriate parameter function t. Update procedures require division, whereas searching uses multiplication only.

4.1 Introduction

A *dictionary* is a data structure that stores a subset S of *keys* from a universe U, and supports answering of membership queries of type "Is x in S?", for $x \in U$. The keys come from the universe $U = \{0,1\}^w$ and may be accompanied by *satellite data* which can be retrieved in case $x \in S$. A dictionary is said to be *dynamic* if it also supports updates of S through insertions and deletions of elements. Otherwise, the dictionary is *static*. The size of set S is standardly denoted by n.

There are several characteristics that make distinction between various kinds of dictionaries, the most important being the amount of space required by the dictionary, the time needed for performing a query, and the time spent on an update or construction of the whole dictionary. We consider only dictionaries with realistic space usage of $O(n)$ registers. We put more priority to search times rather than update times. In some situations updates occur infrequently

compared to queries so having a dictionary with asymmetric query and update times can be useful.

Our model of computation is the *word RAM* model with a standard instruction set, resembling the primitive instructions of the language C. Multiplication and division are included in the instruction set. We adopt the unit-cost model, where all native instructions and memory accesses have cost 1, i.e. their completion takes one unit of time. A usual assumption for RAM dictionaries is that the elements of U fit in one machine word. Contents of a word may be interpreted either as an integer from $\{0, \ldots, 2^w - 1\}$ or as a bit string from $\{0, 1\}^w$.

Algorithms involved in construction of a dictionary may be randomized — they require a source of random bits and their time bounds are either expectations or hold with high probability. Randomized dictionaries reached a stage of high development and theoretically there is little left to be improved. On the other hand, our knowledge about the possibilities of deterministic dictionaries is rather incomplete. It is of theoretical interest to examine performance limits of deterministic dictionaries. A sufficiently simple and efficient deterministic structure would be of significant practical interest as well. Faster and simpler deterministic methods might also help to produce simpler randomized schemes which use few random bits, by assigning some subproblems to deterministic procedures. Yet, it is still hard to imagine a deterministic dictionary with good performance in both theoretical and practical terms. The present work is also primarily interesting on theoretical and conceptual levels.

Many of the modern data structures involve algorithms that are *weakly nonuniform*. Namely, an algorithm is called weakly nonuniform if it assumes availability of certain constants which depend (only) on the word length w. Such constants may be thought of as being computed at "compile time". Some efficient dictionaries [HMP01, Pag00] even require constants that are not known to be computable in polynomial time; this may be problematic even for moderate word sizes. In (theoretical) case of extremely large word sizes, virtually all nonuniform dictionaries give rise to non-negligible costs of preparing the special constants. We give efficient dictionary realizations with uniform algorithms; the code does not require even knowing the value of w. (Yet, access to such simple and natural constants should be allowed in the model; in reality, at least the compiler needs to know the value of w.) A single type of structure is used for the entire range of w (relative to n); there is no splitting into different cases and accordingly applying different types of structures.

4.1.1 Related work

A seminal work by Fredman, Komlós, and Szemerédi [FKS84] showed that in the static case it is possible to construct a linear space dictionary with constant lookup time for arbitrary word sizes (no assumptions about relative values of w and n). This dictionary implementation is known as the *FKS scheme*. Besides a randomized version with expected $O(n)$ construction time, they also gave a deterministic construction algorithm with a running time of $O(n^3 w)$. A bottleneck was choosing of appropriate hash functions. Any *universal* family of hash functions [CW79] contains functions suitable for use in the FKS scheme. Ra-

man [Ram96] devised a deterministic algorithm for finding good functions from a certain universal family, with a running time of $O(n^2 w)$; this implies the same time bound for construction of the FKS dictionary. For $w = n^{\Omega(1)}$, an efficient static dictionary can be built in time $O(n)$ on a sorted sequence of keys. This follows from a generalization of the fusion trees of Fredman and Willard [FW93], and it was observed by Hagerup [Hag98b]. Alon and Naor [AN96] derandomized a variant of the FKS scheme, achieving construction time $O(nw(\log n)^4)$. However, their lookup operation takes time $O(w/\log n)$.

Hagerup, Miltersen and Pagh [HMP01] gave an efficient dictionary with constant lookup time and $O(n \log n)$ construction time. There is a significant nonuniformity in their method. A motivation for the work of this chapter was to achieve a similar performance in dynamic settings, while minimizing the issue of nonuniformity. Later work [Ruž08a, Ruž09] surpassed the performance of the dictionary from [HMP01]. The structures from [Ruž08a, Ruž09] are meant to be used when $w \leq n^{O(1)}$; the case of very large words is left to fusion trees, which means there is a weak nonuniformity.

Allowing randomization, the FKS scheme can be dynamized to support updates in amortized expected constant time [DKM$^+$94]. The lower bound result in the same paper states that a deterministic dynamic dictionary, based on *pure* hashing schemes, with worst-case lookup time of $t(n)$ must have amortized insertion time of $\Omega(t(n) \cdot n^{1/t(n)})$. A standard dynamization technique [OvL81] applied to the static dictionary from [HMP01] yields a similar type of trade-offs: lookups in time $O(t(n))$, insertions in time $O(n^{1/t(n)})$, and deletions in time $O(\log n)$; all time bounds are worst-case, and the parameter function t must satisfy $t(n) = O(\sqrt{\log n})$. A different combination, queries in time $O((\log \log n)^2 / \log \log \log n)$ and updates in time $O((\log n \log \log n)^2)$ is due to Pagh [Pag00].

Among the dictionaries with equal times for all operations, the best result is $O(\sqrt{\log n / \log \log n})$ time per operation; it uses the data structure of Beame and Fich [BF02] with the dynamization result of Andersson and Thorup [AT00]. This data structure also supports neighbour queries.

Andersson et al. [AMRT96] showed that a unit-cost RAM that allows linear space dictionaries with constant query time must have an instruction of circuit depth $\Omega(\log w / \log \log w)$. This matches the circuit depth of multiplication and division. Some work has been done on minimizing query time on RAMs with weaker instruction sets. Most notably, for the AC^0 instruction set, there is tight bound of $\Theta(\sqrt{\log n / \log \log n})$ on query time [AMRT96].

4.1.2 Overview of our contributions

The primitive type of hash functions in our techniques is $h_a(x) = \lfloor r \cdot \mathrm{frac}(ax) \rfloor$. These *multiplicative hash functions* have been known for decades [Knu73]. One concrete family of functions of this type (see (4.2)) was shown to be universal in [DHKP97]. Raman's algorithm selects "good" hash functions from this universal family in bit-by-bit fashion using the method of conditional probabilities.

We do not use the concept of universality directly and we take a bit unusual approach in investigating properties of multiplicative functions by observ-

ing their real extensions. In Section 4.2 we give a characterization of function parameters not causing a collision between a pair of keys, using periodic real sets. It will be easy to observe that only the difference between two keys decides whether they collide for a certain function. We will estimate the measures of the sets of multipliers which produce "few" collisions on a given set of differences. Since we ultimately work with discrete classes of functions, it will be shown that appropriate members of a certain discrete class can be found. Members of our discrete class can be represented using $w + O(\log n)$ bits, which is little more than $w - 1$ bits needed for functions related to the algorithm of Raman.

We describe two different function selection algorithms in Section 4.3. The first one is more general and it accepts as input a special subset of all the differences between the keys from S. The size of this subset affects the running times and the "quality" of the returned function. When the subset is maximal, the algorithm can find both functions usable at the first and at the second levels of the FKS scheme in time $O(n^2 \log n)$. This completely removes the dependency on w present in Raman's algorithm, which was the fastest method for finding a suitable hash function with a description of $O(w)$ bits and an unconstrained range. The scheme of Hagerup et al. [HMP01] includes a faster algorithm for finding appropriate functions with range $\Omega(n^2)$. A small drawback of our algorithm is the use of division. We may make a theoretically interesting remark that the algorithm can be trivially adapted to the *real RAM* model to work with real numbers of arbitrary precision, using only arithmetic operations and comparisons of real numbers. This is not a feature of any other known hashing method.

The second algorithm is specialized for finding perfect hash functions with a polynomial range. It has a running time of $O(nw \log^2 n)$. The range of the function can be tuned, and it influences a constant factor hidden in the big-oh. The range can be as low as n^4 with use of division, and n^7 without use of division. After performing the universe reduction by a chosen function, a dictionary for polynomial-size universes may be used to finish the construction of a static dictionary; see Chapter 3 or [HMP01, Sect. 4].

In Section 4.4 we present a parameterized family of static and dynamic dictionaries based on our new (general) algorithm for choosing a good hash function. In the static case the parameter $\tau : \mathbb{N} \to \mathbb{N}$ should be a nondecreasing function computable in time and space $O(n)$, and satisfying $\tau(n) = O(\sqrt{\log n})$. We attain the combination of search time $O(\tau(n))$ and construction time $O(n^{1+1/\tau(n)})$. For the dynamic dictionaries we add the constraint $\tau(2n) = O(\tau(n))$, and the trade-off is a search time of $O(\tau(n))$ and updates in amortized time $O(n^{1/\tau(n)})$. Our dynamic dictionaries match the result of Hagerup, Miltersen and Pagh, with the exception that our update bounds are amortized instead of worst-case. Our data structures are organized according to a multi-level hashing scheme, which is more natural and rational to dynamize than the static dictionary from [HMP01]. That is why the gap between the static versions is (almost) closed in the dynamic case. Our dictionaries comply with the conditions of the lower bound result from [DKM$^+$94], and the time bounds of our dictionaries match the lower bounds.

Notation

In the whole chapter, S denotes the set that is to be stored, n represents the size of S, and w is the word size of the machine. We denote the elements of S as $\{x_1, x_2, \ldots, x_n\}$. The universe is $U = \{0, \ldots, 2^w - 1\}$, except in Section 4.2.1 where we observe how to efficiently evaluate the hash functions on long keys. The variables r, a, m, and s will also retain their interpretation throughout the text after they get introduced. Logarithms are normally with base 2.

The subset of a set X which contains elements satisfying some property ϕ will usually be written as $\{x \in X \ : \ \phi(x, \ldots)\}$ instead of $\{x \in X \mid \phi(x, \ldots)\}$. The reason is greater clarity, because there will often be absolute values or set cardinality operators in such expressions.

By Q_v we denote $\{u/2^v \ : \ 0 < u < 2^v, \ u \text{ integer}\}$. The set Q_v can be roughly regarded as the set of numbers from $(0, 1)$ given in precision 2^{-v}.

Unlike a single letter m, $m(X)$ denotes Lebesgue measure of a set X. Only simple sets (i.e. finite unions and intersections of intervals) will appear, so thorough knowledge of the theory of Lebesgue measure is not necessary for comprehension of the chapter. The complement of a set X is denoted by X^c.

4.2 Family of Functions

Multiplicative hashing families can generally be regarded as families of type

$$\mathcal{H}_A = \{h_a(x) = \lfloor r \cdot \text{frac}(ax) \rfloor \ : \ a \in A\} \tag{4.1}$$

where $\text{frac}(x)$ denotes $x - \lfloor x \rfloor$. Each member of $\mathcal{H}_A$ maps U to $\{0, \ldots, r - 1\}$. Functions of this type have been used as an old hashing heuristic known as "the multiplication method". The universal family (shown in [DHKP97])

$$\{x \mapsto (ax \bmod 2^w) \text{ div } 2^{w-v} \ : \ a \in U, a \text{ odd}\} \tag{4.2}$$

is of type (4.1). Allowing a to be wider than w bits, the same holds for

$$\{x \mapsto \lfloor (kx \bmod p) \cdot r/p \rfloor \ : \ k \in U, \ p \text{ fixed prime}, \ p > 2^w\} \ ,$$

a variation of the well-known family from [FKS84].

Our dictionaries use functions from $\mathcal{H}_{Q_v}$, where $v = w + O(\log n)$. It will be shown that the discrete families mentioned previously inherit good distribution of elements from their real extension $\mathcal{H}_{\mathbb{R}} = \{h_a : \mathbb{R} \to \mathbf{Z}_r \mid a \in \mathbb{R}\}$. The following simple result is the base of our construction methods.

Lemma 4.2.1. *If $x \neq y$ and*

$$a \in \bigcup_{k \in \mathbf{Z}} \left(\frac{1}{|x - y|} \left(k + \frac{1}{r} \right), \frac{1}{|x - y|} \left(k + 1 - \frac{1}{r} \right) \right) \tag{4.3}$$

then $h_a(x) \neq h_a(y)$, for $h_a \in \mathcal{H}_{\mathbb{R}}$.

Proof. It is easy to see that (4.3) is equivalent to $\operatorname{frac}(a \cdot |x - y|) \in \left(\frac{1}{r}, 1 - \frac{1}{r}\right)$. In case $x > y$ we have

$$
\begin{aligned}
\operatorname{frac}(a \cdot |x - y|) &= \operatorname{frac}(ax - ay) \\
&= \operatorname{frac}(n_1 + \operatorname{frac}(ax) - (n_2 + \operatorname{frac}(ay))) .
\end{aligned}
$$

If $\operatorname{frac}(ax) > \operatorname{frac}(ay)$ then it follows that $\operatorname{frac}(ax) - \operatorname{frac}(ay) > 1/r$. Otherwise, when $\operatorname{frac}(ax) < \operatorname{frac}(ay)$, from $\operatorname{frac}(ax - ay) < 1 - \frac{1}{r}$ it follows that $\operatorname{frac}(ay) - \operatorname{frac}(ax) > 1/r$. The case $y < x$ makes no difference. We conclude that (4.3) implies $|\operatorname{frac}(ax) - \operatorname{frac}(ay)| > 1/r$, which is sufficient for $h_a(x) \neq h_a(y)$ to be true. $\square$

The condition (4.3) is not necessary for avoiding a collision between x and y. Yet, the set of multipliers which certainly cause the collision is not much smaller. We may freely focus on multipliers from $(0, 1)$ because $h_a(x) = h_{a+1}(x)$. Clearly, only the difference between keys x and y affects the set in (4.3). Henceforth, for $d \in U \setminus \{0\}$ we define sets A_d as

$$
A_d = \bigcup_{k=0}^{d-1} \left(\frac{k + \frac{1}{r}}{d}, \ \frac{k + 1 - \frac{1}{r}}{d} \right) .
$$

Sets A_d have a nice periodic structure which will be exploited later.

Let $D = \{d_k\}_{k=1}^s$ be a multiset of elements from $U \setminus \{0\}$. The elements of D are interpreted as differences between some keys. With m being a (fixed) positive integer, we define the set

$$
B = \{a \in (0, 1) \ : \ |\{d \in D \ : \ a \in A_d^c\}| < m\} . \tag{4.4}
$$

Throughout the chapter, m will retain the interpretation of bounding the "allowed" number of collisions. The following relation is obvious.

$$
B \subset \{a \in (0, 1) \ : \ |\{d \in D \ : \ (\exists x \in U)(h_a(x) = h_a(x + d))\}| < m\}
$$

The next lemma shows a bound on the measure of the set of "unsuitable" multipliers, which are not guaranteed to cause less than m collisions. The claim will be referred to in Section 4.3. An interesting corollary is also derived from it.

Lemma 4.2.2. *Let $D = \{d_k\}_{k=1}^s$ be a given multiset of differences, let B be defined as in (4.4), and (a_1, a_2) be an interval within the set $(0, 1)$. Then,*

$$
m\left(B^c \cap (a_1, a_2)\right) < \frac{1}{m} \sum_{k=1}^s m\left(A_{d_k}^c \cap (a_1, a_2)\right) . \tag{4.5}
$$

Proof. Let $K_d : [0, 1] \to \{0, 1\}$ be the characteristic function of set A_d^c. For a fixed a, the number of elements $d \in D$ such that $a \in A_d^c$ is $\sum_{k=1}^s K_{d_k}(a)$. We use more common Riemann integral because it is equal to Lebesgue integral in this case:

$$
\int_{a_1}^{a_2} \sum_{k=1}^s K_{d_k}(t) \, \mathrm{d}t = \sum_{k=1}^s \int_{a_1}^{a_2} K_{d_k}(t) \, \mathrm{d}t = \sum_{k=1}^s m\left(A_{d_k}^c \cap (a_1, a_2)\right) .
$$

The measure of the set $\{a \in (a_1, a_2) \ : \ \sum_{k=1}^{s} K_{d_k}(a) \geq m\}$ can be at most $\frac{1}{m} \sum_{k=1}^{s} m\big(A_{d_k}^c \cap (a_1, a_2)\big)$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

Corollary 4.2.3. *Let $\widehat{C}$ be the set of all a from $(0,1)$ such that the number of pairs $\{x,y\} \in \binom{S}{2}$ for which $h_a(x) = h_a(y)$ is less than m. Then $\widehat{C}$ has a measure of at least $1 - \frac{n(n-1)}{rm}$.*

Proof. Observe that for any d, $m(A_d^c) = 2/r$. Setting $a_1 = 0$, $a_2 = 1$, and the multiset $D = \{|x - y| \ : \ x, y \in S, \ x \neq y\}$, the claim follows from Lemma 4.2.2 and inequality $m(\widehat{C}) \geq 1 - m(B^c)$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

For example, choosing $m = \Theta(n)$ yields functions usable at the first level of the FKS scheme. If the range is set to n and $m = 2n$, then more than "half" of the functions from $\mathcal{H}_{\mathbb{R}}$ would be usable. The corollary demonstrates that the "goodness" of the mentioned families is not *essentially* due to some number-theoretic properties; of course, the way a discrete subfamily of $\mathcal{H}_{\mathbb{R}}$ is chosen does matter.

A set A_d^c consists of disjoint and evenly spaced intervals. We call them support intervals because they form the support of the set's characteristic function. Ends of support intervals are generally *not* in Q_v, where the value of v will be completely determined by Theorem 4.3.1, and it satisfies $v = w + O(\log n) = O(w)$. But our algorithm needs to calculate some measures with bounded precision arithmetic. From now, when referring to *computation* of some measure, the following redefinition of the set A_d^c will be implicitly assumed: for each support interval, its left end is rounded up, and its right end is rounded down to a number in Q_v. Some care must be taken: for example, the bounds of a support interval cannot be determined by adding $1/d$ (which is also rounded) to the bounds of the previous interval. The set derived from A_d^c by rounding its support intervals still contains all unsuitable a's from our discrete set. The parameter v will be large enough so that no rounded support interval becomes empty. Therefore, calculating rounded measures will not violate the correctness of the algorithm given in Theorem 4.3.1.

Finding rounded interval bounds is simpler when $1/r$ can be represented using w bits in the standard encoding. Since r is assumed to be an integer, that is true only when r is a power of two. We restrict the possible ranges of the functions to powers of two, for simplicity.

Lemma 4.2.4. *Given $d \in U \setminus \{0\}$ and $b, c \in Q_v$, let p be the number of support intervals of the set A_d^c which intersect (b, c). The measure of $(b, c) \cap A_d^c$ can be computed in $O(p)$ time with use of division.*

Proof. Let support intervals of A_d^c be $[b_k, c_k]$, where $0 \leq k \leq d$ and b_k, c_k are values rounded in the specified way. For a particular k, the values of b_k, c_k are computed by dividing $k \pm \frac{1}{r}$ by d, producing the first v bits of the result and the information whether the remainder is nonzero. The latter is used in rounding towards infinity when computing b_k. The division takes constant time as the divisor fits in a machine word, the dividend has $2w$ bits, and $O(w)$ bits of the result are computed.

71

Let $i = \max\{k \mid b_k < b\}$ and $j = \min\{k \mid c_k > c\}$; they can be determined in constant time — we use division by $1/d$, which amounts to multiplication. If $i = j$ the result is $c - b$. Otherwise, we calculate the lengths of $(b,c) \cap [b_i, c_i]$ and $(b,c) \cap [b_j, c_j]$. To their sum (which may be zero) we add the lengths of the intermediate support intervals and get the measure. Endpoints of at most $p + 2$ intervals are computed, with each computation taking constant time. $\quad\square$

With the chosen policy of rounding and computation, no numerical errors appear. With a more flexible way of rounding interval endpoints and allowing small numerical error, it would be possible to compute the measure of $(b,c) \cap A_d^c$ in $O(1)$ time. Numerical errors would then have to accounted for in the proof of Theorem 4.3.1. However, the simple method of Lemma 4.2.4 will be used only in situations where $p \leq 2$, and thus is quite sufficient.

Lemma 4.2.5. *Let the set B be defined as in (4.4). If $c - b \leq 2^{-w - \log r}$, then $B \cap (b,c)$ consists of at most m different intervals.*

Proof. Only one support interval from each set A_d^c can have nonempty intersection with (b,c) and it cannot be completely inside (b,c) because it has greater length. Let $(c_k)_{k=1}^p$, $p \leq \binom{n}{2}$, be the increasing sequence of the right ends of support intervals that have nonempty intersection with (b,c). Also let $q = \max\{0, p - m + 1\}$, and $c_0 = b$. Obviously, $(b, c_q]$ is not in B. Suppose that an interval $[a_1, a_2]$ is not in B and that (a_2, a_3) is. The number of collisions decreases at a_2 and this can happen only at one of the points c_{q+i}, $0 \leq i < m$. Thus, there are at most m left boundary points of $B \cap (b,c)$, so the set consists of at most m different intervals. $\quad\square$

The next result is used in the running-time analysis for the algorithm of Theorem 4.3.1.

Lemma 4.2.6. *For any $\alpha > 1$, the set $\{|x - y| \;:\; x, y \in S, \; x \neq y\}$ may be partitioned into at most $n\lceil \log_\alpha n \rceil$ disjoint classes $D_1, D_2, \ldots, D_N$, where every class D_i has the following property: if d_1, d_2 are arbitrary elements of D_i then $d_1 < 3\alpha d_2$.*

Proof. Let $(x_1, x_2, \ldots, x_n)$ be the increasing sequence of keys from S. We prove a slightly stronger claim by induction: for all k, $2 \leq k \leq n$, the set $\{|x_i - x_j| \;:\; i < j \leq k\}$ may be partitioned into $k\lceil \log_\alpha n \rceil$ classes such that the elements of each class are within a factor of $\alpha(1 + \frac{1}{n-1})^{k-2}$ from each other. The case $k = 2$ is trivial.

By the inductive hypothesis, the set $\{|x_i - x_j| \;:\; i < j \leq k\}$ has a partition of size $k\lceil \log_\alpha n \rceil$. Let $d = x_{k+1} - x_k$; d is the smallest difference in $\{|x_{k+1} - x_j| \;:\; j \leq k\}$. Define

$$D_i^{k+1} = \left\{ x_{k+1} - x_j \;:\; j \leq k, \; \alpha^i d \leq x_{k+1} - x_j < \alpha^{i+1}d \right\}, \quad 0 \leq i < \left\lceil \frac{\log n}{\log \alpha} \right\rceil.$$

Add to the partition all the sets D_i^{k+1}, $0 \leq i < \lceil \log_\alpha n \rceil$. The remaining differences induced by x_{k+1} will be distributed to the classes formed in earlier steps.

Let $x_{k+1} - x_j$ be such a difference; then $x_{k+1} - x_j \geq nd$. Also $x_k - x_j \geq (n-1)d$, and

$$\frac{x_{k+1} - x_j}{x_k - x_j} = \frac{x_k - x_j + d}{x_k - x_j} = 1 + \frac{d}{x_k - x_j} \leq 1 + \frac{1}{n-1} \ .$$

We put $x_{k+1} - x_j$ in the class of element $x_k - x_j$. It is easy to verify that after all the differences are put into appropriate classes, the elements of each class are within a factor of $\alpha(1 + \frac{1}{n-1})^{k+1-2}$ from each other. When $k + 1 = n$ we have that $\alpha(1 + \frac{1}{n-1})^{n-2} < \alpha e$, which completes the proof. $\square$

4.2.1 Evaluation on multi-word keys

Suppose that elements of the universe have a length of Lw bits, and suppose that $\log r + \log L < w$. The latter could be easily generalized to $\log r + \log L = O(w)$, which is a reasonable assumption. Let $x = x_{L-1} \ldots x_1 x_0$ and $a = a_{-1} a_{-2} \ldots a_{-L}$ be a key and a multiplier represented as vectors of w-bit words. Define the functions

$$\hat{g}_a(x) = \sum_{i=0}^{L-2} (x_i \cdot a_{-i-1} + x_i \cdot a_{-i-2} \text{ div } 2^w) + x_{L-1} \cdot a_{-L} \ ,$$

and $g_a(x) = (\hat{g}_a(x) \bmod 2^w) \text{ div } 2^{w-\log r}$. Evaluation of function g_a takes time $O(L)$. The value of $g_a(x)$ is either equal to $h_a(x)$ or equal to $h_a(x) - 1$. The size of the longest chain of colliding elements for the function g_a is at most twice the size of the longest chain for the function h_a. The longest chains play a key role in the construction algorithms of the dictionaries of Section 4.4.

4.3 Finding a Good Function

4.3.1 General algorithm

Let $(x_1, x_2, \ldots, x_n)$ be the increasing sequence of keys from S. The main algorithm, described in Theorem 4.3.1, operates on a multiset of differences, denote it by D, which is a subset of the multiset $\{|x - y|\}_{x,y \in S}$. If D could be an arbitrary subset of $\{|x - y|\}_{x,y \in S}$, superlinear space would generally be required to represent it. However, dictionaries described in Section 4.4 require only subsets of the form

$$D = \bigcup_{i=1}^{n-1} \bigcup_{j=1}^{n_i} \{x_{i+j} - x_i\} \ , \tag{4.6}$$

where $(n_i)_{i=1}^{n-1}$ is a sequence of positive integers such that $n_i \leq n-i$. The multiset D is entirely specified by the sequence (n_i) and the sorted list of keys.

Theorem 4.3.1. *Let D be a multiset of differences given by (4.6), and let $s = |D|$. Define $v = w + \lceil \log m \rceil + \lceil \log r \rceil + 3$. If $rm \geq 6s$,[1] we can find a multiplier*

---

[1] A theorem with requirement $\frac{2s}{rm} \leq \nu < 1$, for a constant ν, could similarly be proved. Only constants hidden in the time bound and the bit length of a would be affected.

$a \in Q_v \cap B$ *(B is defined in (4.4)) with a uniform deterministic algorithm that uses space $O(n + m)$ and runs in time*

$$O(s(\log n + \log r) + n \log n \log m + n \log_r n \log^2 m) \ . \qquad (4.7)$$

Proof. On start the search space is the interval $(0, 1)$. The algorithm works in stages, with every stage selecting a smaller subinterval in which to continue the search for a. In most stages the newly selected interval will be a half of the previously selected interval. In some stages the newly selected interval may be a very small subinterval of the previously selected interval. When the active interval gets sufficiently small, the algorithm explicitly determines the intersection of B and that small interval.

In the $(p + 1)$st step the working interval is denoted by $(a_1^{(p)}, a_2^{(p)})$; the superscripts will be omitted when referring to actions in a single step. In every step we observe a partition of the set of differences into three classes. For the $(p + 1)$st step we write $D = D_{\mathrm{big}}^{(p+1)} \cup D_{\mathrm{mid}}^{(p+1)} \cup D_{\mathrm{sml}}^{(p+1)}$ where

$$
\begin{aligned}
D_{\mathrm{big}}^{(p+1)} &= \left\{ d \in D \ : \ \frac{2}{rd} \geq a_2^{(p)} - a_1^{(p)} \right\} \ , \\
D_{\mathrm{mid}}^{(p+1)} &= D \setminus (D_{\mathrm{big}}^{(p+1)} \cup D_{\mathrm{sml}}^{(p+1)}) \ , \\
D_{\mathrm{sml}}^{(p+1)} &= \left\{ d \in D \ : \ \frac{2}{d} < a_2^{(p)} - a_1^{(p)} \right\} \ .
\end{aligned}
$$

Smaller d corresponds to bigger support intervals; sizes and periods of support intervals make the key distinction. Again, superscripts will be omitted when sets and values correspond to one stage of the algorithm. For any $d \in D_{\mathrm{mid}}$ the period of the set A_d^c is not smaller than the length of a half of the working interval. Consequently, at most two support intervals of A_d^c may intersect each half, and at most three support intervals may intersect the whole working interval (a_1, a_2).

For now, we consider the case where $D_{\mathrm{mid}} \neq \emptyset$. In this case, the current step of the algorithm chooses a half of the working interval. Let $b = (a_1 + a_2)/2$. The algorithm will estimate $m(B^c \cap (a_1, b))$ and $m(B^c \cap (b, a_2))$, and the interval with lower value (greater estimated intersection with B) wins.

We first describe in what way the differences from D_{big} are handled. For $d \in D_{\mathrm{big}}$, if the set $(a_1, a_2) \cap A_d^c$ is nonempty and $(a_1, a_2) \nsubseteq A_d^c$, then it is completely determined either by the right end of the intersecting support interval, or the left end of the interval. Support intervals that cover (a_1, a_2) are irrelevant to the choices made by the algorithm. For accounting partial intersections, two ordered sequences are stored — one for the right ends and one for the left ends. Observe that we need information only about the last m right ends and the first m left ends. Let b_1 be the mth largest right end and b_2 be the mth smallest left end (if they don't exist take $b_1 = a_1$ and/or $b_2 = a_2$). The elements of $(a_1, b_1]$ and $[b_2, a_2)$ are certainly not in B. We can employ a balanced tree structure to maintain the two ordered sequences. If the number of elements in the tree that stores the right ends grows to $m + 1$, the smallest element is deleted, as it becomes irrelevant, and the value of b_1 gets updated. Similarly, in the tree that holds the left ends the largest element gets deleted and b_2 changes when the

number of elements grows to $m + 1$. We augment the search trees to efficiently support finding the sum and the number of the stored elements smaller than a given value. The change to the trees is simple to realize, and it provides efficient computation of the values $\sum_{d \in D_{\text{big}}} m(A_d^c \cap (b_1, b))$ and $\sum_{d \in D_{\text{big}}} m(A_d^c \cap (b, b_2))$. The structures are updated as p increases.

From the requirements of the theorem and our construction, it will follow that $B \cap (a_1, a_2) \neq \emptyset$; hence, it must be that $b_1 < b_2$. If $b_2 \leq b$ then (a_1, b) is chosen, if $b_1 \geq b$ then (b, a_2) is chosen. Supposing that $b_1 < b < b_2$, we have:

$$
\begin{aligned}
m(B^c \cap (a_1, b)) &= m((a_1, b_1]) + m(B^c \cap (b_1, b)) \\
&\leq b_1 - a_1 + \frac{1}{m} \sum_{d \in D} m(A_d^c \cap (b_1, b)) \quad \text{(According to Lemma 4.2.2)} \\
&= b_1 - a_1 + \frac{1}{m} \Big(\sum_{d \in D_{\text{big}}} m(A_d^c \cap (b_1, b)) + \\
&\qquad \sum_{d \in D_{\text{mid}}} m(A_d^c \cap (b_1, b)) + \sum_{d \in D_{\text{sml}}} m(A_d^c \cap (b_1, b)) \Big) \ .
\end{aligned}
$$

The sum over the elements of D_{big} can be computed in time $O(\log m)$. Assuming that we can efficiently retrieve the elements of D_{mid}, the terms in the middle sum can be computed in constant time per element, as claimed by Lemma 4.2.4. The last sum is approximated by $\frac{1}{m}|D_{\text{sml}}|(b - b_1)\frac{2}{r}$. The absolute error of this estimate of the actual sum (over $d \in D_{\text{sml}}$) can be bounded by the length of one support interval per each member of D_{sml}:

$$
E = \frac{1}{m} \sum_{d \in D_{\text{sml}}} \frac{2}{rd} \ .
$$

We describe the process of retrieving the elements of $D_{\text{mid}}^{(p+1)}$, and the data structures involved in it. For each i from $\{1, 2, \ldots, n-1\}$ such that $(x_{i+n_i} - x_i) \notin D_{\text{big}}$, we store a double (i, j_i), where $j_i = \min\{j \in \{1, \ldots, n_i\} : (x_{i+j} - x_i) \notin D_{\text{big}}\}$. All doubles (i, j_i) such that $(x_{i+j_i} - x_i) \in D_{\text{mid}}$ are stored in an array in arbitrary order. Knowing j_i it is easy to efficiently retrieve all elements from $\{x_{i+j} - x_i\}_{j=1}^{n_i}$ that belong to D_{mid}. All doubles (i, j_i) such that $(x_{i+j_i} - x_i) \in D_{\text{sml}}$ are stored in a priority queue, with the priority of (i, j_i) being $x_{i+j_i} - x_i$. Between the steps $p+1$ and $p+2$ it is checked if some elements should be extracted from the priority queue and added to the array; this will be the case when the corresponding differences enter the set $D_{\text{mid}}^{(p+2)}$. Likewise, it is checked what elements should be removed from the array and inserted into the priority queue; for index i this will be the case if $(x_{i+j_i} - x_i) \in D_{\text{big}}^{(p+2)}$ and $(x_{i+j_i+1} - x_i) \in D_{\text{sml}}^{(p+2)}$. A simple binary heap will suffice for the priority queue.

Let $\mu_1^{(p+1)}$ denote the value produced by summing the obtained values for the three sums (over the three classes of differences). The value $\mu_1^{(p+1)}$ is our estimate of $m(B^c \cap (a_1, b))$. The other interval (b, a_2) is processed analogously to the first one, and the resulting estimate is denoted by $\mu_2^{(p+1)}$. Note that the error bound E is the sum of errors made on both intervals because the intervals are

consecutive. If $\mu_1^{(p+1)} \leq \mu_2^{(p+1)}$ then (a_1, b) is the new active interval; otherwise, (b, a_2) is the new active interval.

We now describe the operation in the case where $D_{\mathrm{mid}}^{(p+1)} = \emptyset$. We prefetch the difference $\bar{d}$ from the top of the priority queue to see its value. This mode of operation that considers only the elements of D_{big} lasts until we get a working interval of length between $1/\bar{d}$ and $2/\bar{d}$. Let k be the number of meaningful right endpoints of support intervals that have nonempty intersection with $(a_1^{(p)}, a_2^{(p)})$; these meaningful right ends are stored in an augmented search tree as described above, and $k \leq m$. Let c be the $\lfloor k/2 \rfloor$th largest such endpoint. Here we set $b = \max\{a_1 + 1/\bar{d},\ \min\{c, a_2 - 1/\bar{d}\}\}$. The two competing intervals (a_1, b) and (b, a_2) may be asymmetric in terms of length, but each of them has a length of at least $1/\bar{d}$ since $a_2 - a_1 > 2/\bar{d}$ from the definition of $D_{\mathrm{sml}}^{(p+1)}$. We decide to select (b, a_2) iff $\frac{\mu_2}{a_2 - b} \leq \frac{\mu_1}{b - a_1}$. Computation of $\mu_{1/2}^{(p+1)}$ takes $O(\log m)$ time since we consider only the elements of D_{big}. We accomplished to either immediately select an appropriate interval of length $1/\bar{d}$, or to prune away $k/2$ right endpoints from the set of meaningful endpoints. Consequently, after at most $\lceil \log m \rceil$ steps we either get a working interval of length between $1/\bar{d}$ and $2/\bar{d}$, or we get a working interval with no right ends of support intervals of elements from D_{big} in its interior. When the latter case happens, we simply select $(a_1,\ a_1 + 1/\bar{d})$ as the next working interval.

The described process of narrowing the search space is conducted until $a_2^{(p)} - a_1^{(p)} \leq 2^{-w - \lceil \log r \rceil}$. We remark that the value of w need not be known to represent value $2^{-w - \lceil \log r \rceil}$ using two words in a standard way. Let q be the final value of p (i.e. the number of the last conducted step). It is $D = D_{\mathrm{big}}^{(q+1)}$, and Lemma 4.2.5 suggests how we can find a suitable point using the search trees if $m(B \cap (a_1^{(q)}, a_2^{(q)}))$ is large enough. We need to prove that $m(B \cap (a_1^{(q)}, a_2^{(q)}))$ will indeed be large enough.

For $p > 0$, let $\mu^{(p)}$ be the measure estimate (made by the algorithm) for the winning interval selected at the end of the pth step. For the initial estimate $\mu^{(0)}$ we take the bound on $m(B^c \cap (0, 1))$ given by Lemma 4.2.2: $\mu^{(0)} = \frac{2s}{rm} \leq 1/3$. Define $s_{p+1} = |D_{\mathrm{sml}}^{(p)} \setminus D_{\mathrm{sml}}^{(p+1)}|$ for $p > 1$, and $s_1 = 0$. By looking at the structure of our measure estimates, it is not hard to see that if in the $(p + 1)$st step $D_{\mathrm{mid}}^{(p+1)} \neq \emptyset$ then

$$\mu_1^{(p+1)} + \mu_2^{(p+1)} \leq \mu^{(p)} + \frac{s_{p+1}}{m} \cdot \frac{2(a_2^{(p)} - a_1^{(p)})}{r} = \mu^{(p)} + \frac{4s_{p+1}}{rm}(a_2^{(p+1)} - a_1^{(p+1)}) \ .$$

The relation is true for $p = 0$ because in the first step the total contribution of the elements from $D_{\mathrm{sml}}^{(1)}$ to $\mu_{1/2}^{(1)}$ is calculated without error; to see this, remark that $D_{\mathrm{big}}^{(1)} = \emptyset$, and thus $b_1 = 0$, $b_2 = 1$. Setting $\mu^{(p+1)} = \min\{\mu_1^{(p+1)},\ \mu_2^{(p+1)}\}$ gives the inequality

$$\mu^{(p+1)} \leq \frac{1}{2}\mu^{(p)} + \frac{2s_{p+1}}{rm}(a_2^{(p+1)} - a_1^{(p+1)}) \ .$$

If in the $(p+1)$st step $D_{\mathrm{mid}}^{(p+1)} = \emptyset$ then it holds that

$$\mu^{(p+1)} \leq \frac{a_2^{(p+1)} - a_1^{(p+1)}}{a_2^{(p)} - a_1^{(p)}} \mu^{(p)} \ ,$$

which can easily be shown by using that $\mu_1^{(p+1)} + \mu_2^{(p+1)} = \mu^{(p)}$ in this case. A simple induction argument shows that in any case

$$\frac{\mu^{(p+1)}}{a_2^{(p+1)} - a_1^{(p+1)}} \leq \frac{1}{3} + \frac{2}{rm} \sum_{i=0}^{p} s_{i+1} \ . \tag{4.8}$$

Since $\sum_{i=0}^{q} s_{i+1} \leq s$, from (4.8) it follows that

$$m(B^c \cap (a_1^{(q)}, a_2^{(q)})) \leq \mu^{(q)} \leq \frac{2}{3} m((a_1^{(q)}, a_2^{(q)})) \ . \tag{4.9}$$

According to Lemma 4.2.5 and (4.9), within $B \cap (a_1^{(q)}, a_2^{(q)})$ there must be an interval of length at least $\frac{1}{3m}(a_2^{(q)} - a_1^{(q)})$. Hence, using the balanced trees, in time $O(m)$ we can find such an interval and within it an element from $Q_{w + \lceil \log r \rceil + \lceil \log m \rceil + 3}$.

It remains to complete the time analysis. When $D_{\mathrm{mid}}^{(p+1)} \neq \emptyset$, the contribution of elements from $D_{\mathrm{mid}}^{(p+1)}$ to $\mu_{1/2}^{(p+1)}$ is computed in time $O(|D_{\mathrm{mid}}^{(p+1)}|)$. We show that an element of D stays in D_{mid} at most $\lceil \log r \rceil$ steps. The element d is in $D_{\mathrm{mid}}^{(p+1)}$ iff

$$\frac{a_2^{(p)} - a_1^{(p)}}{r} \leq \frac{2}{rd} < a_2^{(p)} - a_1^{(p)} \iff \frac{1}{2(a_2^{(p)} - a_1^{(p)})} \geq d > \frac{1}{2r(a_2^{(p)} - a_1^{(p)})} \ . \tag{4.10}$$

Let p_1 be the smallest value of p for which $\frac{1}{2}(a_2^{(p)} - a_1^{(p)})^{-1} \geq d$. Since the working interval is at least halved in each stage, it will be

$$\frac{1}{2r(a_2^{(p_1 + \lceil \log r \rceil)} - a_1^{(p_1 + \lceil \log r \rceil)})} \geq \frac{1}{2(a_2^{(p_1)} - a_1^{(p_1)})} \geq d \ .$$

Therefore, $d \notin D_{\mathrm{mid}}^{(p+1+\lceil \log r \rceil)}$. The cost of using the priority queue can be simply bounded by $O(\log n)$ per element of D. There would be no gain in using a priority queue structure more advanced than the binary heap, because the processing time for a difference while it is in D_{mid} is typically dominant. Each element of D eventually enters D_{big}, and update time per element is $O(\log m) = O(\log n)$ (the parameter m makes sense only if lower than $\binom{n}{2}$). Thus, the total load related to D_{mid} is $O(s(\log n + \log r))$.

The tree structures are used to acquire the contribution of elements from $D_{\mathrm{big}}^{(p+1)}$ to $\mu_{1/2}^{(p+1)}$. The procedure, which takes $O(\log m)$ time, occurs at most c_u times, where c_u is the number of p-s for which $D_{\mathrm{mid}}^{(p+1)} \neq \emptyset$. The tree structures are also used in finding an appropriate subinterval in the case $D_{\mathrm{mid}}^{(p+1)} = \emptyset$. The

total time spent in this part of the algorithm is $O(c_t \log^2 m)$, where c_t is the number of transitions to the state $D_{\text{mid}}^{(p+1)} = \emptyset$.

It remains to determine bounds on c_t and c_u. Set $\alpha = r/6$ and observe the partition of $\{|x - y| \ : \ x, y \in S, \ x \neq y\}$ given by Lemma 4.2.6. This induces a partition of D of the same size of $O(n \log_r n)$. As a consequence of (4.10), when the first element of a certain class (in the partition) enters D_{mid}, it will not leave D_{mid} before the last element of the class enters D_{mid}. It easily follows that the number of p-s for which elements of one class belong to $D_{\text{mid}}^{(p)}$ is $O(\log r)$; hence, $c_u = O(n \log_r n \log r) = O(n \log n)$. Since D_{mid} certainly does not become empty for the period of processing elements from one class, the number of transitions c_t is not greater than the number of classes: $c_t = O(n \log_r n)$.

Putting all the bounds together gives (4.7). Nonconstant space requirements come from the balanced tree structures and the data structures used for retrieving the members of D_{mid}. They need space $O(m)$ and $O(n)$, respectively. $\square$

Usually $\log r = \Theta(\log n)$, so a simple time upper bound is $O(s \log n + n \log^2 n)$. Remark that the algorithm itself is not too complicated, in spite of lengthy justification of its actions. Only classical data structures appear as auxiliary structures in the method; thereby aggregate implementation complexity is not high. The disadvantages of the algorithm are the use of division and high dependence on n. For typical relations of parameters w, m, r the evaluation of the function h_a would require 2-3 multiplications.

4.3.2 Reduction to a polynomial-size universe in time $O(nw \log^2 n)$

In this section we describe a more specialized algorithm for finding a perfect hash function with range polynomial in n. Throughout the section we assume that $m = 1$, because we seek for perfect functions only. Consequently, $B^c = \cup A_d^c$.

When the measure of the set B is rather large, e.g. when $r = \Omega(n^5)$, we may view the problem of deciding bits of a function parameter as that of avoiding very bad choices, rather than finding the best. Intuitively, we expect that calculating a rough approximation of $\mu_{1/2}^{(p)}$ will suffice in this case. The ability to perform more approximative calculations will enable us to reduce implicit redundancy in computation by processing groups of differences at once. First it will be shown how to find measure estimates up to a constant factor higher than those obtained by the algorithm of Theorem 4.3.1. That way, we will be able to reject "terrible" intervals, not choose the better ones. It will be proved that this is good enough for selecting a perfect function with a polynomial range.

Let $F_d = \{k/d \mid 0 \leq k \leq d\}$. A point from F_d belongs to an interval $(b - \nu, b]$ iff

$$b - \nu < \left(b \text{ div } \frac{1}{d}\right) \frac{1}{d} \leq b$$
$$\iff b - \nu < \frac{\lfloor bd \rfloor}{d} \iff bd - \lfloor bd \rfloor < \nu d \ . \tag{4.11}$$

Function frac is a homomorphism from $(R, +, 0)$ to $([0, 1), +_1, 0)$, where $x +_1 y$

represents the operation $\mathrm{frac}(x + y)$. Putting $d = x - y$ in (4.11), we get

$$\mathrm{frac}(b(x - y)) = \mathrm{frac}(bx) -_1 \mathrm{frac}(by) < \nu d \ . \tag{4.12}$$

One of the key ideas in the proof of Theorem 4.3.1 was almost ignoring the elements of D_{sml} in every step. The support intervals of those elements repeat with periods less than $(a_2 - a_1)/2$, which means that at least two support intervals intersect with each half of the working interval. Here, we see that the condition (4.12) makes sense only when $d < 1/\nu$; otherwise there is always a nonempty intersection. In the following we assume $d < 1/\nu$. Transforming the condition (4.12) into an expression over $\mathbb{R}$ only, yields

$$F_d \cap (b - \nu, b] \neq \emptyset \iff \begin{aligned} &\mathrm{frac}(bx) - \nu d < \mathrm{frac}(by) \leq \mathrm{frac}(bx) \\ \vee\ &\mathrm{frac}(bx) + 1 - \nu d < \mathrm{frac}(by) \ . \end{aligned} \tag{4.13}$$

Note that it is not possible for both disjuncts to be true.

A point from F_d belongs to $(a, a + \nu]$ if and only if $(\lfloor ad \rfloor + 1)/d \leq a + \nu$. Proceeding analogously to the above, we get a condition over $\mathbb{R}$:

$$F_d \cap (a, a + \nu] \neq \emptyset \iff \begin{aligned} &\mathrm{frac}(ax) < \mathrm{frac}(ay) \leq \mathrm{frac}(ax) + \nu d \\ \vee\ &\mathrm{frac}(ay) \leq \mathrm{frac}(ax) - 1 + \nu d \ . \end{aligned} \tag{4.14}$$

Suppose that interval (a, b) is fixed; this will later be a half of the working interval in one bit-choosing step. Given a set $\widehat{S} \subset U$, suppose that $x \in U$ satisfies $x > \max \widehat{S}$ and $1/(x - \min \widehat{S}) > b - a$; this ensures that $(\forall y \in \widehat{S}) |F_{x-y} \cap (a, b]| \leq 1$. Define the function

$$\mu_{ab}(\widehat{S}, x) = \sum m(A_{x-y}^c \cap (a, b)) + m\left(\bigcup \left(A_{x-y}^c \cap (a, b) \right) \right) \ , \tag{4.15}$$

where the sum is over all $y \in \widehat{S}$ for which $F_{x-y} \cap (a, b] \neq \emptyset$, and the union is over all $y \in \widehat{S}$ for which $F_{x-y} \cap (a, b] = \emptyset$ (subscript ab will be omitted when the interval is understood). Lebesgue measure is a subaditive function, and thus

$$\mu_{ab}(\widehat{S}, x) \geq m\left(\bigcup_{y \in \widehat{S}} \left(A_{x-y}^c \cap (a, b) \right) \right) \ .$$

Lemma 4.3.2. *Suppose that interval (a, b) is fixed. Let $\widehat{S} \subset U$, with $\hat{n} = |\widehat{S}|$. There is a dynamic data structure storing $\widehat{S}$ and supporting queries of the following type.*

Check whether a given $x \in U$ satisfies: $x > \max \widehat{S}$, $x - \min \widehat{S} < 2(x - \max \widehat{S})$, $1/(x - \min \widehat{S}) > b - a$, and $\frac{2}{r(x - \max \widehat{S})} < b - a$. If it does, then return $\sigma(x)$ such that $\mu(\widehat{S}, x) \leq \sigma(x) < 4\mu(\widehat{S}, x)$.

Updates and queries are performed in $O(\log \hat{n})$ worst-case time.

Proof. We use two instances of a structure supporting rank and neighbour queries, for example an augmented balanced tree. One stores $\{\operatorname{frac}(ay) \mid y \in \widehat{S}\}$, and the other stores $\{\operatorname{frac}(by) \mid y \in \widehat{S}\}$. It is trivial to check whether x is a regular input to a query — all we need are $\min \widehat{S}$ and $\max \widehat{S}$.

Denote the first and the second term in (4.15) by $\mu'(x)$ and $\mu''(x)$ respectively. We first describe how to approximate $\mu'(x)$. Set $\hat{d} = x - \min \widehat{S}$ and $\nu = (b-a)/2$. From (4.13) it follows that

$$F_{x-y} \cap \left(b - \nu \frac{\hat{d}}{x - y}, \, b \right] \neq \emptyset \quad \Longleftrightarrow \quad \operatorname{frac}(bx) - \nu\hat{d} < \operatorname{frac}(by) \leq \operatorname{frac}(bx)$$
$$\vee \quad \operatorname{frac}(bx) + 1 - \nu\hat{d} < \operatorname{frac}(by) \; .$$

The previous equivalence specifies appropriate rank queries which determine

$$\{y \in \widehat{S} \; : \; F_{x-y} \cap (b - \nu\hat{d}/(x - y), \, b] \neq \emptyset\}$$

by two pairs of indices, one for each disjunct. Analogously, from (4.14) we derive rank queries which determine $\{y \in \widehat{S} \; : \; F_{x-y} \cap (a, a + \nu\hat{d}/(x-y)] \neq \emptyset\}$ by another two pairs of indices. Some elements may appear in both sets; finding the union takes constant time. Denote the size of the union by $\bar{n}$. Since $1 \leq \hat{d}/(x - y) < 2$,

$$\bar{n} = \left| \{y \in \widehat{S} \; : \; F_{x-y} \cap (a, b] \neq \emptyset\} \right| \; .$$

Finally, set $\sigma'(x) = \bar{n} \frac{2}{r(x - \max \widehat{S})}$. If all support intervals that intersect with (a, b) lie completely inside (a, b), we would have an upper bound $\sigma'(x) < 2\mu'(x)$. But measures due to partially intersecting intervals may be down to one half of their length. Thus, the upper bound of $\sigma'(x) < 4\mu'(x)$ is tight.

For the value $\mu''(x)$ we have

$$
\begin{aligned}
\mu''(x) &= \max \left(\max \left\{ \lfloor a(x - y) \rfloor \frac{1}{x - y} + \frac{1}{r(x - y)} - a \; : \; y \in \widehat{S} \right\}, \, 0 \right) \\
&\quad + \max \left(\max \left\{ b - \frac{\lfloor b(x - y) \rfloor + 1}{x - y} + \frac{1}{r(x - y)} \; : \; y \in \widehat{S} \right\}, \, 0 \right) \\
&= \max \left(\max \left\{ \frac{1}{x - y} \left(-\operatorname{frac}(a(x - y)) + \frac{1}{r} \right) \; : \; y \in \widehat{S} \right\}, \, 0 \right) \\
&\quad + \max \left(\max \left\{ \frac{1}{x - y} \left(\operatorname{frac}(b(x - y)) - 1 + \frac{1}{r} \right) \; : \; y \in \widehat{S} \right\}, \, 0 \right) \; .
\end{aligned}
$$

We approximate $\mu''(x)$ by

$$
\begin{aligned}
\sigma''(x) &= \max \left(\frac{1}{x - \max \widehat{S}} \left(-\min_{y \in \widehat{S}} \operatorname{frac}(a(x - y)) + \frac{1}{r} \right), \, 0 \right) \\
&\quad + \max \left(\frac{1}{x - \max \widehat{S}} \left(\max_{y \in \widehat{S}} \operatorname{frac}(b(x - y)) - 1 + \frac{1}{r} \right), \, 0 \right) \; .
\end{aligned}
$$

The minimum of $\operatorname{frac}(a(x - y)) = \operatorname{frac}(ax) -_1 \operatorname{frac}(ay)$ can be found as

$$\min \left(\operatorname{frac}(ax) - \operatorname{predecessor}(\operatorname{frac}(ax)), \, \operatorname{frac}(ax) + 1 - \max_{y}(\operatorname{frac}(ay)) \right) \; ,$$

where predecessor(X) returns the largest element smaller than, or equal to X (that is, not strictly smaller). Similarly, the maximum of $\mathrm{frac}(b(x - y))$ equals to

$$\max\left(\mathrm{frac}(bx) + 1 - \mathrm{successor}(\mathrm{frac}(bx)),\ \mathrm{frac}(bx) - \min_y(\mathrm{frac}(by))\right)\ ,$$

with successor(X) returning the smallest element strictly larger than X. It is not hard to realize that $\mu''(x) \le \sigma''(x) \le 2\mu''(x)$, which proves the lemma. $\qquad\square$

Theorem 4.3.3. *Suppose that $r = n^7$, and let $q = w + \lceil \log r \rceil$. In time $O(nw \log^2 n)$ we can deterministically find $a \in Q_{q+1}$ such that h_a is perfect on S. The algorithm is uniform.*

Proof. At the beginning, the elements of S are sorted to get an increasing sequence $(x_i)_{i=1}^n$. The working interval in the $(p+1)$st step is denoted by $(a_1^{(p)}, a_2^{(p)})$; superscripts will be omitted when referring to actions in a single step. Unlike in the algorithm of Theorem 4.3.1, $(a_1^{(p+1)}, a_2^{(p+1)})$ will usually be little smaller than a half of $(a_1^{(p)}, a_2^{(p)})$. It will be shown that $2^{-(p+1)} < a_2^{(p)} - a_1^{(p)} \le 2^{-p}$. Recall that $b = (a_1 + a_2)/2$.

In this algorithm $D = \{|x - y|\ :\ x, y \in S,\ x \ne y\}$, so all the differences, and hence sets A_d^c, are eventually accounted for. The classes of differences in one step are: $D_{\mathrm{big}}^{(p+1)} = \{d \in D\ :\ \frac{2}{rd} \ge \frac{2^{-p}}{4}\}$, $D_{\mathrm{sml}}^{(p+1)} = \{d \in D\ :\ \frac{1}{d} < \frac{2^{-p}}{2}\}$, and $D_{\mathrm{mid}} = D \setminus (D_{\mathrm{big}} \cup D_{\mathrm{sml}})$. Remark that the classes are defined little differently than in Theorem 4.3.1. An action at the end of each step will ensure that $A_d^c \cap (a_1, a_2) = \emptyset$ for every $d \in D_{\mathrm{big}}$.

Selection of intervals proceeds until $p = q$. Any point from $(a_1^{(q)}, a_2^{(q)})$ produces a perfect function, since $D = D_{\mathrm{big}}^{(q+1)}$. We choose a point with the smallest possible bit length, namely, a value from Q_{q+1}.

Define the sets $C_{p+1} = D_{\mathrm{sml}}^{(p)} \setminus D_{\mathrm{sml}}^{(p+1)}$, $p \ge 0$. Set C_{p+1} contains all the differences that entered D_{mid} in the $(p+1)$st step. Let $t = \lceil \log r \rceil - 3$. The number of steps a difference stays in D_{mid} is $t + 1$, which is shown as in the analogous part in Theorem 4.3.1. We may write $D_{\mathrm{mid}}^{(p+1)} = \bigcup_{k=0}^{\min(t,p)} C_{p+1-k}$.

In a single bit-choosing step, for every $i > 1$ and every k such that $0 \le k \le \min\{t, p\}$, we define functions $\mathrm{left}(i, k)$ and $\mathrm{right}(i, k)$, returning indices which satisfy the following:

- $\mathrm{left}(i, k) \le \mathrm{right}(i, k) \le i$;

- If $x_i - x_1 \in C_{p+1-k_1}$ and $k_1 > k$, then $\mathrm{left}(i, k) = \mathrm{right}(i, k) = 0$;

- If $x_i - x_{i-1} \in C_{p+1-k_1}$ and $k_1 < k$ (k_1 may be negative), then $\mathrm{left}(i, k) = \mathrm{right}(i, k) = i$;

- Otherwise, let

$$
\begin{aligned}
\mathrm{left}(i, k) &= \min\{j\ :\ x_i - x_j \in C_{p+1-k_1} \wedge k_1 \ge k\}\ , \\
\mathrm{right}(i, k) &= 1 + \max\{j\ :\ x_i - x_j \in C_{p+1-k_1} \wedge k_1 \le k\}\ .
\end{aligned}
$$

Consequently, $\{x_i - x_j \;:\; \operatorname{left}(i,k) \leq j < \operatorname{right}(i,k)\} \subset C_{p+1-k}$. If $C_{p+1-k} \cap \{x_i - x_j \;:\; j < i,\ \text{fixed } i\} = \emptyset$ then $\operatorname{left}(i,k) = \operatorname{right}(i,k)$.

For $0 \leq k \leq \min\{t,p\}$, we define

$$\eta_1^{(p+1)}(k) = \sum_{i=2}^{n} \mu_{a_1 b}(\{x_{\operatorname{left}(i,k)}, \ldots, x_{\operatorname{right}(i,k)-1}\}, x_i) \ ,$$

and we set $\eta_1^{(p+1)}(k) = 0$ for $k > p$. Now we combine values $\eta_1^{(p+1)}(k)$, defining

$$\eta_1^{(p+1)} = \sum_{k=0}^{t} K^{t-k} \eta_1^{(p+1)}(k) \ , \tag{4.16}$$

with $K = \frac{8}{5}$. Analogous values for the interval (b, a_2) are defined similarly and they are denoted by $\eta_2^{(p+1)}(k)$ and $\eta_2^{(p+1)}$. In every step we compute approximations of η_1 and η_2 — denote them by σ_1 and σ_2 — and the interval with lower value wins. First we describe the computation of $\sigma_{1/2}$, and afterwards we prove the correctness of the algorithm.

The approximations $\sigma_{1/2}$ are calculated in an iteration where i goes from 2 to n. For any k, updating $\operatorname{left}(i,k)$ and $\operatorname{right}(i,k)$ takes total $O(n)$ time, as they may increase only, and checking whether $\operatorname{left}(i,k) = x_j$, for a particular x_j, consumes constant time. Summing the work of updating $\operatorname{left}(i,k)$ and $\operatorname{right}(i,k)$, for all k, gives a load of $O(nt) = O(n \log n)$. The set

$$\{x_{\operatorname{left}(i,0)}, x_{\operatorname{left}(i,0)+1}, \ldots, x_{\operatorname{right}(i,t)-1}\}$$

is kept partitioned into $t + 1$ classes of consecutive elements: the kth class is given by $\{x_{\operatorname{left}(i,k)}, \ldots, x_{\operatorname{right}(i,k)-1}\}$ provided that $\operatorname{left}(i,k) \neq \operatorname{right}(i,k)$, and it is empty otherwise. If y, z are arbitrary elements of a class then $x_i - y < 2(x_i - z)$. Each class is stored in two instances of the data structure of Lemma 4.3.2 — one for the interval (a_1, b) and the other for the interval (b, a_2). Maintaining the properties of the partition as i changes requires movement of elements from one class to another. Every element of S may be moved $t + 1$ times per bit-choosing step, implying $O(n \log^2 n)$ time cost of updating the data structures. For every i, the data structures are queried and the results are added to the partial sums of $\sigma_{1/2}(k)$, the approximations of $\eta_{1/2}(k)$. After final iteration, $\sigma_{1/2}(k)$ are combined as in (4.16) producing σ_1 and σ_2. The sum of query times is $O(n \log^2 n)$ per bit-choosing step.

Without loss of generality, suppose that the first interval wins, that is, $\sigma_1 \leq \sigma_2$. By Lemma 4.3.2:

$$\eta_1 \leq \sigma_1 \leq \sigma_2 < 4\eta_2 \ . \tag{4.17}$$

It will be shown that η_1 (and $\eta_1(t)$, as a result) is small enough to disallow $F_d \cap (a_1, b) \neq \emptyset$, for any $d \in C_{p+1-t}$. The method of Lemma 4.3.2 (the second part of the proof) is used to determine $(a_1^{(p+1)}, a_2^{(p+1)})$ such that: $a_1^{(p+1)} - a_1 + b - a_2^{(p+1)} \leq 2\eta_1(t)$ and $(\forall d \in C_{p+1-t}) \left(A_d^c \cap (a_1^{(p+1)}, a_2^{(p+1)}) = \emptyset \right)$.

The description of the algorithm and time analysis are now completed. We are left to show the correctness of the operation.

From $\eta_1^{(p+1)}(k)+\eta_2^{(p+1)}(k) \leq \eta^{(p)}(k-1)$ and (4.16) we get a relation between $\eta_{1/2}^{(p+1)}$ and $\eta^{(p)}$:

$$\eta_1^{(p+1)} + \eta_2^{(p+1)} \leq K^{-1}\left(\eta^{(p)} - \eta^{(p)}(t)\right) + K^t\left(\eta_1^{(p+1)}(0) + \eta_2^{(p+1)}(0)\right) \ .$$

Eliminating $\eta_2^{(p+1)}$ using (4.17) gives

$$\eta_1^{(p+1)} \leq \frac{4}{5}K^{-1}\left(\eta^{(p)} - \eta^{(p)}(t)\right) + \frac{4}{5}K^t\left(\eta_1^{(p+1)}(0) + \eta_2^{(p+1)}(0)\right) \ . \qquad (4.18)$$

By induction on p we will prove that

$$\frac{\eta^{(p)} - \eta^{(p)}(t)}{a_2^{(p)} - a_1^{(p)}} \leq \left(|D| - |D_{\text{sml}}^{(p)}|\right)\frac{4}{5}K^t\frac{16}{r} - \frac{1}{4}\sum_{i=1}^{p}\frac{2\eta^{(i)}(t)}{\frac{1}{2}(a_2^{(i-1)} - a_1^{(i-1)})} \ , \qquad (4.19)$$

and also that $a_2^{(p)} - a_1^{(p)} > 2^{-(p+1)}$. Before the induction argument, observe a consequence of (4.19).

$$\begin{aligned}\frac{\eta^{(p)} - \eta^{(p)}(t)}{a_2^{(p)} - a_1^{(p)}} &\leq \ |D|\frac{4}{5}K^{\log r - 3}\frac{16}{r} = |D|\frac{4}{5}\left(\frac{r}{8}\right)^{\log\frac{8}{5}}\frac{16}{r} = \frac{25}{8}|D|r^{\log\frac{8}{5}-1} \\ &< \ 2n^2 r^{\log\frac{8}{5}-1} = 2n^2 n^{7(\log\frac{8}{5}-1)} < 2n^{-\frac{1}{4}} \ . \qquad (4.20)\end{aligned}$$

The sum of ratios of cut off regions in (4.19) will be needed to prove the second part of the hypothesis regarding the length of the working intervals.

Before the first step, when $p = 0$, it is $(a_1^{(0)}, a_2^{(0)}) = (0,1)$ and $D_{\text{mid}}^{(0)} = D \cap \{1\}$. Therefore $\eta^{(0)} = K^t\eta^{(0)}(0) = |D_{\text{mid}}^{(0)}|K^t\frac{2}{r}$, and so (4.19) holds for $p = 0$. Suppose that (4.19) is true for some p. From (4.18) we get

$$\frac{\eta_1^{(p+1)}}{b - a_1^{(p)}} \leq \frac{1}{2}\frac{\eta^{(p)} - \eta^{(p)}(t)}{\frac{1}{2}(a_2^{(p)} - a_1^{(p)})} + \frac{4}{5}K^t\frac{\eta_1^{(p+1)}(0) + \eta_2^{(p+1)}(0)}{\frac{1}{2}(a_2^{(p)} - a_1^{(p)})} \ .$$

From the definitions of the sets C_{p+1} and $D_{\text{sml}}^{(p+1)}$, it follows that $2^{-p-1} \leq 1/d < 2^{-p}$, for every $d \in C_{p+1}$. For any $d \in C_{p+1}$ and any interval of length 2^{-p}, denote it by I_p, we may bound $m(A_d^c \cap I_p)$ by the measure of two support intervals (though, for larger d, $A_d^c \cap I_p$ may span across three support intervals, two of which only partially intersect with I_p). Hence, $\eta_1^{(p+1)}(0)+\eta_2^{(p+1)}(0) \leq |C_{p+1}|2\frac{2}{r2^p}$. The inductive hypothesis, including $a_2^{(p)} - a_1^{(p)} > 2^{-(p+1)}$, and the bound on $\eta_1^{(p+1)}(0) + \eta_2^{(p+1)}(0)$ give

$$\begin{aligned}\frac{\eta_1^{(p+1)}}{b - a_1^{(p)}} &\leq \ \left(|D| - |D_{\text{sml}}^{(p)}|\right)\frac{4}{5}K^t\frac{16}{r} - \sum_{i=1}^{p}\frac{\eta^{(i)}(t)}{a_2^{(i-1)} - a_1^{(i-1)}} + \frac{4}{5}K^t|C_{p+1}|4\frac{4}{r} \\ &= \ \left(|D| - |D_{\text{sml}}^{(p+1)}|\right)\frac{4}{5}K^t\frac{16}{r} - \sum_{i=1}^{p}\frac{\eta^{(i)}(t)}{a_2^{(i-1)} - a_1^{(i-1)}} \ . \qquad (4.21)\end{aligned}$$

Denoting $X = \frac{2\eta^{(p+1)}(t)}{b - a_1^{(p)}} = \frac{2\eta^{(p+1)}(t)}{\frac{1}{2}(a_2^{(p)} - a_1^{(p)})}$, we have

$$\frac{\eta_1^{(p+1)} - \eta_1^{(p+1)}(t)}{a_2^{(p+1)} - a_1^{(p+1)}} \leq \frac{\eta_1^{(p+1)} - \eta_1^{(p+1)}(t)}{b - a_1^{(p)} - 2\eta_1^{(p+1)}(t)} = \frac{\eta_1^{(p+1)} - \eta_1^{(p+1)}(t)}{b - a_1^{(p)}} \frac{1}{1 - X} \ .$$

Taylor's expansion of the function $f(x) = (1 + x)^{-1}$ with Lagrange's remainder is $f(x) = 1 - x + x^2(1 + \Theta x)^{-3}$, and it holds for $|x| < 1$. For $-\frac{1}{4} < x \leq 0$, an upper bound is $f(x) \leq 1 + 2|x|$. As a result of (4.20) and the relation $\eta^{(p+1)}(t) < K^{-1}\left(\eta^{(p)} - \eta^{(p)}(t)\right)$, it follows $X < 5n^{-\frac{1}{4}} < \frac{1}{4}$ for $n = \Omega(1)$. Therefore, $(1 - X)^{-1} \leq 1 + 2X$. Since also $\frac{\eta_1^{(p+1)} - \eta_1^{(p+1)}(t)}{b - a_1^{(p)}} < \frac{1}{8}$ by (4.21) and (4.20), we obtain

$$\frac{\eta_1^{(p+1)} - \eta_1^{(p+1)}(t)}{a_2^{(p+1)} - a_1^{(p+1)}} \leq \frac{\eta_1^{(p+1)} - \eta_1^{(p+1)}(t)}{b - a_1^{(p)}} + \frac{1}{4}X \ .$$

Finally,

$$\frac{\eta_1^{(p+1)} - \eta_1^{(p+1)}(t)}{a_2^{(p+1)} - a_1^{(p+1)}} \leq \frac{\eta_1^{(p+1)}}{b - a_1^{(p)}} - \frac{1}{4}X \ ,$$

which together with (4.21) proves the first part of the claim for $p + 1$.

It remains to show that $a_2^{(p+1)} - a_1^{(p+1)} > 2^{-(p+2)}$. At the end of the $(p+1)$st step, the working interval is shrank in such a way that

$$a_2^{(p+1)} - a_1^{(p+1)} \geq \frac{1}{2}\left(a_2^{(p)} - a_1^{(p)}\right)\left(1 - \frac{2\eta^{(p+1)}(t)}{\frac{1}{2}(a_2^{(p)} - a_1^{(p)})}\right) \ . \qquad (4.22)$$

Inequalities (4.19) and (4.20) imply

$$\sum_{i=1}^{p+1} \frac{2\eta^{(i)}(t)}{\frac{1}{2}(a_2^{(i-1)} - a_1^{(i-1)})} < 8n^{-\frac{1}{4}} \ .$$

The minimum of function $f(x_1, x_2, \ldots, x_{p+1}) = (1 - x_1)(1 - x_2)\cdots(1 - x_{p+1})$ under condition $x_1 + \cdots + x_{p+1} = c$ is attained for $x_i = \frac{c}{p+1}$, $1 \leq i \leq p + 1$. Thus, after unrolling (4.22) we get

$$a_2^{(p+1)} - a_1^{(p+1)} \geq 2^{-(p+1)}\left(1 - \frac{8}{n^{\frac{1}{4}}(p+1)}\right)^{p+1} > 2^{-(p+2)} \ .$$

$\square$

We make observations on how to change the range of the universe reduction function and how to avoid the use of division. Lemma 4.3.2 can be restated so that query regularity conditions include $x - \min \widehat{S} < \alpha(x - \max \widehat{S})$, for $1 < \alpha \leq 2$, instead of $x - \min \widehat{S} < 2(x - \max \widehat{S})$. Then $\sigma(x)$, the query result, will satisfy $\mu(\widehat{S}, x) \leq \sigma(x) < 2\alpha\mu(\widehat{S}, x)$. The main algorithm is modified so that every set C_{p+1-k} is broken into $1/\log \alpha$ sets which are stored in separate data structures. Instead of (4.17), we have $\eta_1 \leq \sigma_1 \leq \sigma_2 < 2\alpha\eta_2$. Then, we may set $K = 2\frac{2\alpha}{2\alpha+1}$.

With obvious modifications, the rest of the analysis is preserved. For example, letting $\alpha = 1.2$ and acting as in (4.20), it may be checked that $r = n^4$ is large enough.

The only place where division is needed is the computation of $\frac{1}{x - \max \widehat{S}}$ in the algorithm of Lemma 4.3.2. Approximating the value by a nearby power of 2 avoids division but makes the result of the query less precise. Namely, in this case the result satisfies $\mu(\widehat{S}, x) \leq \sigma(x) < 4\alpha\mu(\widehat{S}, x)$.

4.4 Uniform Dictionaries

Theorem 4.4.1. *Let $\tau : \mathbb{N} \to \mathbb{N}$ be a nondecreasing function computable in time and space $O(n)$, and satisfying $\tau(n) = O(\sqrt{\log n})$. There exists a linear space static deterministic dictionary that has a query time of $O(\tau(n))$ and construction time $O(n^{1+1/\tau(n)})$.*

Proof. We employ a multi-level hashing scheme. Levels of the tree are constructed top-down and parameters for hash functions differ from level to level. Let $t = 3\tau(n)$. There will be at most t levels. A lookup consists of t evaluations of hash functions and following pointers to lower levels. The last two levels are resolved using the two-level FKS scheme, and function parameters, that is, s, r, and m, are set accordingly. We describe the construction of the remaining levels.

The FKS scheme is never applied immediately at the first level, since $t \geq 3$. For the top-level hash function we set $n_i = \min\{2\lceil n^{1/t}\rceil, n - i\}$, $1 \leq i \leq n$ (recall the structure of the set D given in (4.6)), and let the range be $r = 3n/t$. Also, we allow $m = 4t\lceil n^{1/t}\rceil$ collisions, thereby satisfying the condition of Theorem 4.3.1 that $\frac{2s}{rm} \leq 1/3$. The algorithm described in Theorem 4.3.1 returns a value a. A time bound for the first level is $O(n^{1+1/t}\log n)$.

Define $P_i = \{j \in \{i + 1, \ldots, i + n_i\} : h_a(x_i) = h_a(x_j)\}$; it holds that $\sum_{i=1}^{n-1}|P_i| < m$. Suppose that $h_a(x_i) = y$, for some $i \leq n - 2\lceil n^{1/t}\rceil$. At least $2n^{1/t} - |P_i|$ keys cannot hash to slot y. Take $x_j \in S$, such that $j \leq n - 2\lceil n^{1/t}\rceil$, $i \notin P_j$, and $h_a(x_j) = y$. Because of the structure of D, *another* $2n^{1/t} - |P_j|$ keys cannot hash to slot y. From the set $\{x_k, \ldots, x_n\}$, with $k = n - 2\lceil n^{1/t}\rceil + 1$, at most $1 + \sum_{i=k}^{n-1}|P_i|$ elements can hash to slot y. Through an induction argument we get an upper bound on the longest chain: $\frac{1}{2}n^{1-1/t} + m$. From $3 \leq t(n) \leq O(\sqrt{\log n})$ it follows that $m = o(n^{1-1/t(n)})$. Assuming that n is sufficiently large, we may write $m < \frac{1}{2}n^{1-1/t}$. Hence, the length of the longest chain is less than $n^{1-1/t}$.

In general, let N be the size of a bucket at the kth level, where $1 \leq k \leq t - 2$ (the first level bucket is the whole set S). Suppose that $N > \log^2 n$; otherwise we choose to apply the FKS scheme. Let $u(k) = \left(1 - \frac{k-1}{t}\right)t$. The procedure for selecting a hash function for the node of that bucket takes parameters

$$s = 2N\left\lceil N^{1/u(k)}\right\rceil, \quad r = \frac{3N}{t}, \quad m = 4t\left\lceil N^{1/u(k)}\right\rceil .$$

As $k \leq t - 2$, we have $u(k) \geq 3$. Therefore $m = O(t \cdot N^{1/3})$. The assumption $N > \log^2 n$ provides $t(n) = o(N^{1/3})$. Then $m = o(N^{2/3})$ and we can bound

the longest chain by $N^{1-1/u(k)}$, when n (and hence N) is large enough. The construction procedure is performed recursively.

By induction, the number of elements in a node at the kth level does not exceed $n^{1-(k-1)/t}$. As a result, at most $n^{2/t}$ elements can be mapped to a node at level $t-1$. Note that $\log^2 n = o(n^{2/O(\sqrt{\log n})})$.

For the function $f(x) = x^{1+1/u(k)}$ it holds that $f(x_1 + x_2 + \cdots + x_k) > f(x_1) + f(x_2) + \cdots + f(x_k)$. Thus, the constructions at the kth level will take the most time if all nonempty slots have the same size that matches the upper bound on bucket size. This bounds the time of constructing the kth level, excluding the FKS constructions, by

$$n^{\frac{k-1}{t}} O\left(\left(n^{1-\frac{k-1}{t}}\right)^{1+1/u(k)} \log n\right) = O\left(n^{1+1/t} \log n\right) .$$

The total time spent on the FKS constructions is at most

$$n^{1-2/t} O\left(\left(n^{2t}\right)^2 \log n\right) = O\left(n^{1+2/t} \log n\right) .$$

The factor of $\log n$ can be subdued by a factor of $n^{1/t}$. Adding the times for all the levels and substituting t back to $3\tau(n)$ gives the claimed construction time.

The setting for r at each node makes the total space consumption stay linear in n. □

The described static dictionary is dynamized naturally because data is stored in a tree structure.

Theorem 4.4.2. *Let $\tau : \mathbb{N} \to \mathbb{N}$ be a nondecreasing function computable in time and space $O(n)$, and satisfying $\tau(n) = O(\sqrt{\log n})$ and $\tau(2n) = O(\tau(n))$. There exists a linear space dynamic deterministic dictionary with query time $O(\tau(n))$ and update time $O(n^{1/\tau(n)})$. The bound on lookup time is worst case, while the bound for update times is amortized.*

Proof. A rebuilding scheme of varying locality is used. A rebuilding initiated at some node causes the reconstruction of the whole subtree rooted at that node. Here N will denote the number of elements of S in a subtree after the last reconstruction and k again represents the level of a particular node. The value of k will not change as long as the node exists. The construction method of Theorem 4.4.1 is slightly modified by setting $t = 5\tau(2n)$ and doubling the function parameters s and m at each (non FKS) node. This causes the longest chain to be at most $\frac{1}{2}N^{1-1/u(k)}$ immediately after a reconstruction. The value of parameter t occurs in update procedures and it changes only on global rebuildings. A global rebuilding is the one which is triggered at the top level node. Due to constraint $\tau(2n) = O(\tau(n))$ there are at most $t = O(\tau(n))$ levels of the tree. Also if n gets halved during updates of S there will still be $O(\tau(n))$ levels.

Rebuilding of the bottom two nodes on any path takes place whenever insertion causes a collision and no reconstruction at a higher level is triggered. Local rebuilding of a (non FKS) subtree is performed in two cases: when updates cause the longest chain at the root of the subtree to grow to $N^{1-1/u(k)}$, or when the number of elements drops below $N/2$. The former is aimed at preserving the

claimed search time, while the latter is aimed at keeping the space usage linear. A global rebuilding is performed when those conditions are satisfied at the root node *or* when the size of S doubles compared to the value on the previous global rebuilding.

For analysis, assume that only insertions are performed; it will be clear that this is worse than a case of mixed insertions and deletions. Suppose that the number of elements in a subtree that is to be rebuilt is δN. At least $\frac{1}{2}N^{1-1/u(k)}$ elements must have been inserted for reconstruction to occur; the number can be this small only when all new elements are "pumped" to the same slot in the root of the subtree and that slot had contained a chain of maximum length upon the last rebuilding. Thus, $\delta \geq 1 + \frac{1}{2}N^{-1/u(k)}$.

Let $\bar{n}$ be a value such that $\bar{n}^{1-(k-1)/t} = \delta N$; it will be smaller than n when the subtree is smaller than the maximal allowed size for a bucket at the kth level. Then buckets in the reconstructed subtree may have size not more than $\bar{n}^{1-(j-1)/t}$, where j is the level number in the global tree, $j \geq k$ (the bound is less tight than in Theorem 4.4.1 because we neglect a factor of $1/2$ per each level; yet, this is ultimately insignificant). A time upper bound for the jth level, excluding the FKS constructions, is

$$\delta N \; / \; \bar{n}^{1-\frac{j-1}{t}} \; O\left(\left(\bar{n}^{1-\frac{j-1}{t}} \right)^{1+1/u(j)} \log n \right) = O\left(\delta N \bar{n}^{1/t} \log n \right) \; .$$

The time spent on the FKS constructions is at most

$$O\left(\delta N \bar{n}^{2/t} \log n \right) = \delta N \; O\left(n^{2/t} \log n \right) \; .$$

The latter dominates over the sum of t former times. The cost assigned to each of $\delta N - N$ new elements is

$$\frac{\delta}{\delta - 1} \; O\left(n^{3/t} \right) = N^{1/u(k)} \; O\left(n^{3/t} \right) = O\left(\bar{n}^{1/t} n^{3/t} \right) = O\left(n^{4/t} \right) \; .$$

Each inserted key may be regarded as a "new element" in rebuildings up to t times. The total amortized cost of an update becomes $O(t \cdot n^{4/t}) = O(n^{5/t})$, which finishes the proof since $t > 5\tau(n)$ at any time before the next global rebuilding. $\square$

Acknowledgments

Chapter 5

Linear Probing with Constant Independence

Abstract

Hashing with linear probing dates back to the 1950s, and is among the most studied algorithms. In recent years it has become one of the most important hash table organizations since it uses the cache of modern computers very well. Unfortunately, previous analyses rely either on complicated and space consuming hash functions, or on the unrealistic assumption of free access to a hash function with random and independent function values. Already Carter and Wegman, in their seminal paper on universal hashing, raised the question of extending their analysis to linear probing. However, we show that linear probing using a pairwise independent family may have expected *logarithmic* cost per operation. On the positive side, we show that 5-wise independence is enough to ensure constant expected time per operation. This resolves the question of finding a space and time efficient hash function that provably ensures good performance for linear probing.

5.1 Introduction

Hashing with linear probing is perhaps the simplest algorithm for storing and accessing a set of keys that obtains nontrivial performance. Given a hash function h, a key x is inserted in an array by searching for the first vacant array position in the sequence $h(x), h(x)+1, h(x)+2, \ldots$ (Here, addition is modulo r, the size of the array.) Retrieval of a key proceeds similarly, until either the key is found, or a vacant position is encountered, in which case the key is not present in the data structure. Deletions can be performed by moving keys backward in the probe sequence in a greedy fashion (ensuring that no key x is moved to before $h(x)$), until no such move is possible (when a vacant array position is encountered).

Linear probing dates back to 1954, but was first analyzed by Knuth in a 1963 memorandum [Knu63] now considered to be the birth of the area of analysis of algorithms [Pe98]. Knuth's analysis, as well as most of the work that has since gone into understanding the properties of linear probing, is based on the assump-

tion that h has uniformly distributed and independent function values. In 1977, Carter and Wegman's notion of universal hashing [CW79] initiated a new era in the design of hashing algorithms, where explicit and efficient ways of choosing hash functions replaced the unrealistic assumption of complete randomness. In their seminal paper, Carter and Wegman state it as an open problem to "Extend the analysis to [...] double hashing and open addressing."[1]

5.1.1 Previous results using limited randomness

The first analysis of linear probing relying only on limited randomness was given by Siegel and Schmidt in [SS90, SS95]. Specifically, they show that $O(\log n)$-wise independence is sufficient to achieve essentially the same performance as in the fully random case. (We use n to denote the number of keys inserted into the hash table.) Another paper by Siegel [Sie04] shows that evaluation of a hash function from a $O(\log n)$-wise independent family requires time $\Omega(\log n)$ unless the space used to describe the function is $n^{\Omega(1)}$. A family of functions is given that achieves space usage n^ϵ and constant time evaluation of functions, for any $\epsilon > 0$. However, this result is only of theoretical interest since the associated constants are very large (and growing exponentially with $1/\epsilon$).

A potentially more practical method is the "split and share" technique described in [DW05]. It can be used to achieve characteristics similar to those of linear probing, still using space n^ϵ, for any given constant $\epsilon > 0$. The idea is to split the set of keys into many subsets of roughly the same size, and simulate full randomness on each part. Thus, the resulting solution would be a *collection* of linear probing hash tables.

A significant drawback of both methods above, besides a large number of instructions for function evaluation, is the use of random accesses to the hash function description. The strength of linear probing is that for many practical parameters, almost all lookups will incur only a single cache miss. Performing random accesses while computing the hash function value may destroy this advantage.

According to our knowledge, the first paper in analysis of algorithms where exactly 5-wise independence appeared was [KR93]. They study a version of Quicksort that uses a 5-wise independent pseudorandom number generator.

5.1.2 Our results

We show in this chapter that linear probing using a pairwise independent family may have expected *logarithmic* cost per operation. Specifically, we resolve the open problem of Carter and Wegman by showing that linear probing insertion of n keys in a table of size $2n$ using a function of the form $x \mapsto ((ax + b) \bmod p) \bmod 2n$, where $p = 4n + 1$ is prime and we randomly choose $a \in [p] \backslash \{0\}$ and $b \in [p]$, requires $\Omega(n \log n)$ insertion steps in expectation for a worst

---

[1]Nowadays the term "open addressing" refers to any hashing scheme where the data structure is an array containing only keys and empty locations. However, Knuth used the term to refer to linear probing in [Knu63], and since it is mentioned here together with the double hashing probe sequence, we believe that it refers to linear probing.

case insertion sequence (chosen independently of a and b). Since the total insertion cost equals the total cost of looking up all keys, the expected average time to look up a key in the resulting hash table is $\Omega(\log n)$. The main observation behind the proof is that if a is the multiplicative inverse (modulo p) of a small integer m, then inserting a certain set that consists of two intervals has expected cost $\Omega(n^2/m)$.

On the positive side, we show that *5-wise independence* is enough to ensure constant expected time per operation, for load factor $\alpha \overset{\text{def}}{=} n/r$ bounded away from 1. Our proof is based on a new way of bounding the cost of linear probing operations, by counting intervals in which "many" probe sequences start. When beginning this work, our first observation was that a key x can be placed in location $h(x) + l \bmod r$ only if there is an interval $I \ni h(x)$ where $|I| \geq l$ and there are $|I|$ keys from S with hash value in I. A slightly stronger fact is shown in Lemma 5.4.1. Since the expected number of hash values in an interval I is $\alpha|I|$, long such intervals are "rare" if the hash function exhibits sufficiently high independence.

Our analysis gives a bound of $O(\frac{1}{(1-\alpha)^{13/6}})$ expected time per operation at load factor α. This implies a bound of $O(\frac{1}{(1-\alpha)^{7/6}})$ expected time on average for successful searches. These bounds are a factor $\Omega(\frac{1}{(1-\alpha)^{1/6}})$ higher than for linear probing with full independence. (The exponent can be made arbitrarily close to zero by increasing the independence of the hash function.)

The gap to the fully random case vanishes if we slightly change the probe sequence to

$$h(x), \ h(x) \oplus 1 \, h(x) \oplus 2, \ h(x) \oplus 3, \ldots$$

where $\oplus$ denotes bitwise exclusive or. Also, the range r should be a power of 2. This probe sequence is arguably even more cache friendly than classical linear probing if we assume that memory block boundaries are at powers of 2. In fact, in Section 5.5 we analyze a slightly more general class of open addressing methods called *blocked probing*, which also includes a special kind of bidirectional linear probing. For this class we get the same dependence on α (up to constant factors) as for full independence, again using only 5-wise independent hash functions. A particularly precise analysis of successful searches is conducted, showing that the expected number of probes made during a search for a random element in the table is less than $1 + \frac{2}{1-\alpha}$.

In Section 5.6 we describe an alternative to linear probing that preserves the basic property that all memory accesses of an operation are within a small interval, but improves the expected lookup time exponentially to $O(\log(\frac{1}{1-\alpha}))$.

5.1.3   Significance

Several recent experimental studies [BMQ98, HL05, PR04] have found linear probing to be the fastest hash table organization for moderate load factors (30-70%). While linear probing operations are known to require more instructions than those of other open addressing methods, the fact that they access an interval of array entries means that linear probing works very well with modern architectures for which sequential access is much faster than random access (as-

suming that the keys we are accessing are each significantly smaller than a cache line, or a disk block, etc.). However, the hash functions used to implement linear probing in practice are heuristics, and there is no known theoretical guarantee on their performance. Since linear probing is particularly sensitive to a bad choice of hash function, Heileman and Luo [HL05] advice *against* linear probing for general-purpose use. Our results imply that simple and efficient hash functions, whose description can be stored in CPU registers, can be used to give provably good expected performance.

The work on linear probing given in this chapter has been built upon by other researchers in designing hash tables with additional considerations. Blelloch and Golovin [BG07] described a linear probing hash table implementation that is *strongly history independent*. Thorup [Tho09] studied how to get *efficient* compositions of hash functions for linear probing when the domain of keys is complex, like the set of variable-length strings.

5.2 Preliminaries

5.2.1 Notation and definitions

Let $[x] \stackrel{\text{def}}{=} \{0, 1, \ldots, x-1\}$. Throughout this chapter S denotes a subset of some universe U, and h will denote a function from U to $R \stackrel{\text{def}}{=} [r]$. We denote the elements of S by $\{x_1, x_2, \ldots, x_n\}$, and refer to the elements of S as *keys*. We let $n \stackrel{\text{def}}{=} |S|$, and $\alpha \stackrel{\text{def}}{=} n/r$.

A family $\mathcal{H}$ of functions from U to R is k-wise independent if for any k distinct elements $x_1, \ldots, x_k \in U$ and h chosen uniformly at random from $\mathcal{H}$, the random variables $h(x_1), \ldots, h(x_k)$ are independent[2]. We refer to the variable

$$\bar{\alpha}_{\mathcal{H}} \stackrel{\text{def}}{=} n \max_{x \in U, \rho \in R} \Pr_{h \in \mathcal{H}}\{h(x) = \rho\}$$

as the *maximum load* of $\mathcal{H}$. When the hash function family in question is understood from the context, we omit the subscript of $\bar{\alpha}$. If $\mathcal{H}$ distributes hash function values of all elements of U uniformly on R, we will have $\bar{\alpha} = \alpha$, and in general $\bar{\alpha} \geq \alpha$.

For $Q \subseteq R$ we introduce notation for the "translated set"

$$a + Q \stackrel{\text{def}}{=} \{(a + y) \bmod r \mid y \in Q\} \ .$$

An *interval* (modulo r) is a set of the form $a + [b]$, for integers a and b. When we write $[a - b, a)$ this interval represents the set $a - 1 - [b]$. We will later use sets of the form $h(x) + Q$, for a fixed x and with Q being an interval.

5.2.2 Hash function families

Carter and Wegman [WC81] observed that the family of degree $k-1$ polynomials in any finite field is k-wise independent. Specifically, for any prime p we may

---

[2]We note that in some papers, the notion of k-wise independence is stronger in that it is required that function values are uniformly distributed in R. However, some interesting k-wise independent families have a slightly nonuniform distribution, and we will provide analysis for such families as well.

use the field defined by arithmetic modulo p to get a family of functions from $[p]$ to $[p]$ where a function can be evaluated in time $O(k)$ on a RAM, assuming that addition and multiplication modulo p can be performed in constant time. To obtain a smaller range $R = [r]$ we may map integers in $[p]$ down to R by a modulo r operation. This of course preserves independence, but the family is now only close to uniform. Specifically, the maximum load $\bar{\alpha}$ for this family is in the range $[\alpha, (1 + r/p)\alpha]$. By choosing p much larger than r we can make $\bar{\alpha}$ arbitrarily close to α.

A recently proposed k-wise independent family of Thorup and Zhang [TZ04] has uniformly distributed function values in $[r]$, and thus $\bar{\alpha} = \alpha$. From a theoretical perspective (ignoring constant factors) it is inferior to Siegel's highly independent family [Sie04], since the evaluation time depends on k and the space usage is the same (though the dependence of ϵ is better). We mention it here because it is the first construction that makes k-wise independence truly competitive with popular heuristics, for small $k > 3$, in terms of evaluation time. In practice, the space usage can be kept so small that it does not matter. The construction for 4-wise independence has been shown to be particularly efficient. Though this is not stated in [TZ04], it is not hard to verify that the same construction in fact gives 5-wise independence, and thus our analysis will apply.

5.2.3 A probabilistic lemma

Here we state a lemma that is essential for our upper bound results, described in Section 5.4. It gives an upper bound on the probability that an interval around a particular hash function value contains the hash function values of "many" keys. The proof is similar to the proof of [KRS90, Lemma 4.19].

Lemma 5.2.1. *Let $S \subseteq U$ be a set of size n, and $\mathcal{H}$ a 5-wise independent family of functions from U to R with maximum load at most $\bar{\alpha} < 1$. If h is chosen uniformly at random from $\mathcal{H}$, then for any $Q \subset R$ of size q, and any fixed $x \in U \setminus S$,*

$$\Pr\left\{\left|\{y \in S \;:\; h(y) \in (h(x) + Q)\}\right| \geq \bar{\alpha}q + d\right\} < \frac{4\bar{\alpha}q^2}{d^4} \;.$$

Proof. Denote by A the event that $\left|\{y \in S \;:\; h(y) \in (h(x) + Q)\}\right| \geq \bar{\alpha}q + d$. We will show a stronger statement, namely that the same upper bound holds for the conditional probability $\Pr\{A \mid h(x) = \rho\}$, for any $\rho \in R$. Notice that the subfamily $\{h \in \mathcal{H} \mid h(x) = \rho\}$ is 4-wise independent on $U \setminus \{x\}$, and that the distribution of function values is identical to the distribution when h is chosen from $\mathcal{H}$. The statement of the lemma will then follow from

$$\Pr(A) = \sum_{\rho \in R} \Pr\{h(x) = \rho\}\Pr\{A \mid h(x) = \rho\} < r \cdot \frac{1}{r}\frac{4\bar{\alpha}q^2}{d^4} \;.$$

Let $p_i \stackrel{\text{def}}{=} \Pr\{h(x_i) \in (h(x) + Q)\}$, and consider the random variables

$$X_i \stackrel{\text{def}}{=} \begin{cases} 1 - p_i, & \text{if } h(x_i) \in h(x) + Q \\ -p_i, & \text{otherwise} \end{cases} \;.$$

Let $X \overset{\text{def}}{=} \sum_i X_i$ and observe that

$$\left|\{y \in S \ : \ h(y) \in (h(x) + Q)\}\right| = X + \sum_i p_i \leq X + \bar{\alpha} q \ .$$

The last inequality above is by the definition of maximum load. So to prove the lemma it suffices to bound $\Pr\{X \geq d\}$. We will use the 4th moment inequality

$$\Pr\{X \geq d\} \leq \mathrm{E}(X^4)/d^4 \ .$$

Clearly, $\mathrm{E}(X_i) = 0$ for any i, and the variables $X_1, \ldots, X_n$ are 4-wise independent. Therefore we have $\mathrm{E}(X_{i_1} X_{i_2} X_{i_3} X_{i_4}) = 0$ unless $i_1 = i_2 = i_3 = i_4$ or (i_1, i_2, i_3, i_4) contains 2 numbers, both of them exactly twice. This means that

$$\begin{aligned}
\mathrm{E}(X^4) &= \sum_{1 \leq i_1, i_2, i_3, i_4 \leq n} \mathrm{E}(X_{i1} X_{i2} X_{i3} X_{i4}) \\
&= \sum_{1 \leq i \leq n} \mathrm{E}(X_i^4) + \sum_{1 \leq i < j \leq n} \binom{4}{2} \mathrm{E}(X_i^2) \mathrm{E}(X_j^2).
\end{aligned}$$

The first sum can be bounded as follows:

$$\begin{aligned}
\sum_i \mathrm{E}(X_i^4) &= \sum_i (p_i (1 - p_i)^4 + (1 - p_i) p_i^4) \\
&= \sum_i p_i (1 - p_i)((1 - p_i)^3 + p_i^3) \\
&< \sum_i p_i \leq \bar{\alpha} q \ .
\end{aligned}$$

The second sum is:

$$\begin{aligned}
\sum_{1 \leq i < j \leq n} 6(p_i (1 - p_i))(p_j (1 - p_j)) &< 3 \sum_{1 \leq i,j \leq n} p_i p_j \\
&= 3(\sum_i p_i)^2 \leq 3(\bar{\alpha} q)^2 \ .
\end{aligned}$$

In conclusion we have

$$\Pr\{X \geq d\} \leq \mathrm{E}(X^4)/d^4 < \frac{3(\bar{\alpha} q)^2 + \bar{\alpha} q}{d^4} < \frac{4 \bar{\alpha} q^2}{d^4} \ ,$$

finishing the proof. □

5.3 Pairwise independence

In this section we show that pairwise independence is not sufficient to ensure good performance for linear probing: Logarithmic time per operation is needed for a worst-case set. This complements our upper bounds for 5-wise (and higher) independence. We will consider two pairwise independent families: The first one is a very commonly used hash function family. The latter family is similar to the first, except that we have ensured function values to be uniformly distributed in R. To lower bound the cost of linear probing we use the following lemma.

Lemma 5.3.1. *Suppose a set S of n keys is inserted in a linear probing hash table of size $r > n$. Let $\{S_j\}_{j=1}^{\ell}$ be any partition of S such that for every set S_j the set $I_j \stackrel{def}{=} h(S_j)$ is an interval (modulo r), and $|I_j| \leq r/2$. Then the total number of steps to perform the insertions is at least*

$$\sum_{1 \leq j_1 < j_2 \leq \ell} |I_{j_1} \cap I_{j_2}|^2/2 \ .$$

Proof. We proceed by induction on ℓ. Since the number of insertion steps is independent of the order of insertions [Knu98, p. 538], we may assume that the insertions corresponding to S_ℓ occur last and in left-to-right order of hash values. By the induction hypothesis, the total number of steps to do all preceding insertions is at least

$$\sum_{1 \leq j_1 < j_2 \leq \ell-1} |I_{j_1} \cap I_{j_2}|^2/2 \ .$$

For $1 \leq j_1, j_2 \leq \ell$ let $S_{j_1 j_2}$ denote the set of keys from S_{j_1} that have probe sequences starting in I_{j_2}, i.e $S_{j_1 j_2} = \{x \in S_{j_1} \mid h(x) \in I_{j_2}\}$. For any $x \in S_{\ell j}$ the insertion of x will pass all the elements of $S_{j\ell}$ "after h(x)", i.e., whose hash value is in $h(x) + [r/2]$. This means that at least $|I_j \cap I_\ell|^2/2$ steps are used during the insertions of the keys from S_ℓ to pass locations occupied by keys of S_j. Summing over all $j < \ell$ and adding to the bound from the induction hypothesis yields the desired result. $\square$

5.3.1 Linear congruential hash functions

We first consider the following family of functions, introduced by Carter and Wegman [CW79] as a first example of a universal family of hash functions:

$$\mathcal{H}(p,r) \stackrel{def}{=} \{x \mapsto ((ax+b) \bmod p) \bmod r \mid 0 < a < p, 0 \leq b < p\}$$

where p is any prime number and $r \leq p$ is any integer. Functions in $\mathcal{H}(p,r)$ map integers of $[p]$ to $[r]$.

Theorem 5.3.2. *For $r = \lceil p/2 \rceil$ there exists a set $S \subseteq [p]$, $|S| \leq r/2$, such that the expected cost of inserting the keys of S in a linear probing hash table of size r using a hash function chosen uniformly at random from $\mathcal{H}(p,r)$ is $\Omega(r \log r)$.*

Proof. We give a randomized construction of S, and show that when choosing h at random from $\mathcal{H}(p,r)$ the expected total insertion cost for the keys of S is $\Omega(r \log r)$. This implies the existence of a fixed set S with at least the same expectation for random $h \in \mathcal{H}(p,r)$. Specifically, we partition $[p]$ into 8 intervals $U_1, \ldots, U_8$, such that $\bigcup_i U_i = [p]$ and $r/4 \geq |U_i| \geq r/4 - 1$ for $i = 1, \ldots, 8$, and let S be the union of two of the sets $U_1, \ldots, U_8$ chosen at random (without replacement). Note that $|S| \leq r/2$, as required.

Consider a particular function $h \in \mathcal{H}(p,r)$ and the associated values of a and b. Let $\hat{h}(x) \stackrel{def}{=} (ax+b) \bmod p$, and let m denote the unique integer in $[p]$ such that $am \bmod p = 1$ (i.e., $m = a^{-1}$ in $GF(p)$). Since $\hat{h}$ is a permutation on $[p]$, the sets $\hat{h}(U_i)$, $i = 1, \ldots, 8$, are disjoint. We note that for any x, $\hat{h}(x+m) =$

$(\hat{h}(x) + 1) \bmod p$. Thus, for any k, $\hat{h}(\{x, x+m, x+2m, \ldots, x+km\})$ is an interval (modulo p) of length $k+1$. This implies that for all i there exists a set $\hat{L}_i$ of m disjoint intervals such that $\hat{h}(U_i) = \bigcup_{I \in \hat{L}_i} I$. Similarly, for all i there exists a set L_i of at most $m+1$ intervals (not necessarily disjoint) such that we have the multiset equality $h(U_i) = \bigcup_{I \in L_i} I$. Since all intervals in $\bigcup_i \hat{L}_i$ are disjoint and their sizes differ by at most 1, an interval in $\bigcup_i L_i$ can intersect at most two other intervals in $\bigcup_i L_i$. We now consider two cases:

1. Suppose there is some i such that

$$\sum_{I_1, I_2 \in L_i, I_1 \neq I_2} |I_1 \cap I_2| \geq r/16 \ . \tag{5.1}$$

   With constant probability it holds that $U_i \subseteq S$. We apply Lemma 5.3.1 on the set U_i and on a partition of U_i that corresponds to the interval collection L_i. The lemma gives us a lower bound of

$$\sum_{I_1, I_2 \in L_i, I_1 \neq I_2} |I_1 \cap I_2|^2 / 2 \tag{5.2}$$

   on the number of probes made during all insertions. This sum is minimized if all nonzero intersections have the same size. Suppose that there are $k = O(m)$ nonzero intersections. According to (5.1) the equal size of intersections would have to be $\Omega(r/k)$. Therefore the sum in (5.2) is $\Omega(r^2/k) = \Omega(r^2/m)$.

2. Now suppose that for all i,

$$\sum_{I_1, I_2 \in L_i, I_1 \neq I_2} |I_1 \cap I_2| < r/16 \ .$$

   Note that any value in $[r-1]$ is contained in exactly two intervals of $\bigcup_i L_i$. By the assumption, the number of values that occur in two intervals from the same collection L_i, for any i, is less than $8 \cdot r/16 = r/2$. Thus there exist i_1, i_2, $i_1 \neq i_2$, such that $|h(U_{i_1}) \cap h(U_{i_2})| = \Omega(r)$. With constant probability we have that $S = U_{i_1} \cup U_{i_2}$. We now apply Lemma 5.3.1. Consider just the terms in the sum of the form $|I_1 \cap I_2|^2 / 2$, where $I_1 \in L_{i_1}$ and $I_2 \in L_{i_2}$. As before, this sum is minimized if all $O(m)$ intersections have the same size, and we derive an $\Omega(r^2/m)$ lower bound on the number of insertion steps.

For a random $h \in \mathcal{H}(p, r)$, m is uniformly distributed in $\{1, \ldots, p\}$ (the mapping $a \mapsto a^{-1}$ is a permutation of $\{1, \ldots, p\}$). This means that the expected total insertion cost is:

$$\Omega\left(\frac{1}{p} \sum_{m=1}^{p} r^2/m\right) = \Omega\left(\frac{r^2}{p} \log p\right) = \Omega(r \log r) \ .$$

$\square$

5.3.2 Family with uniform distribution

One might wonder if the lower bound shown in the previous section also holds if the hash function values are uniformly distributed in R. We slightly modify $\mathcal{H}(p, r)$ to remain pairwise independent and also have uniformly distributed function values. Let $\hat{p} \stackrel{\text{def}}{=} \lceil p/r \rceil\, r$, and define:

$$g(y, \hat{y}) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \hat{y} & \text{if } \hat{y} \geq p \\ y & \text{otherwise} \end{array} \right. \quad .$$

For a vector v let v_i denote the $i + 1$st component (indexes starting with zero). We define:

$$\mathcal{H}^*(p, r) \stackrel{\text{def}}{=} \{x \mapsto g((ax + b) \bmod p, v_x) \bmod r \mid 0 \leq a < p,\, 0 \leq b < p,\, v \in [\hat{p}]^p\}$$

Lemma 5.3.3 (Pairwise independence)**.** *For any pair of distinct values $x_1, x_2 \in [p]$, and any $y_1, y_2 \in [r]$, if h is chosen uniformly at random from $\mathcal{H}^*(p, r)$, then*

$$\Pr\{h(x_1) = y_1 \wedge h(x_2) = y_2\} = 1/r^2 \quad .$$

Proof. We will show something stronger than claimed, namely that the family

$$\mathcal{H}^{**} = \{x \mapsto g((ax + b) \bmod p, v_x) \mid 0 \leq a < p,\, 0 \leq b < p,\, v \in [\hat{p}]^p\}$$

is pairwise independent and has function values uniformly distributed in $[\hat{p}]$. Since r divides $\hat{p}$ this will imply the lemma. Pick any pair of distinct values $x_1, x_2 \in [p]$, and consider a random function $h \in \mathcal{H}^{**}$. Clearly, v_{x_1} and v_{x_2} are uniform in $[\hat{p}]$ and independent. We note as in [CW79] that for any $y_1', y_2' \in [p]$ there is exactly one choice of a and b that makes $(ax_1 + b) \bmod p = y_1'$ and $(ax_2 + b) \bmod p = y_2'$. This is because the matrix $\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \end{pmatrix}$ is invertible. As a consequence, $(ax_1 + b) \bmod p$ and $(ax_2 + b) \bmod p$ are uniform in $[p]$ and independent. We can think of the definition of $h(x)$ as follows: The value is v_x unless $v_x \in [p]$, in which case we substitute v_x for another random value in $[p]$, namely $(ax + b) \bmod p$. It follows that hash function values are uniformly distributed, and pairwise independent. $\square$

Corollary 5.3.4. *Theorem 5.3.2 holds also if we replace $\mathcal{H}(p, r)$ by $\mathcal{H}^*(p, r)$. In particular, pairwise independence with uniformly distributed function values is not a sufficient condition for linear probing to have expected constant cost per operation.*

Proof. Consider the parameters a, b, and v of a random function in $\mathcal{H}^*(p, r)$. Since $r = \lceil p/2 \rceil$ we have $\hat{p} = p + 1$, and $(p/\hat{p})^p > 1/4$. Therefore, with constant probability it holds that $a \neq 0$ and $v \in [p]^p$. Restricted to functions satisfying this, the family $\mathcal{H}^*(p, r)$ is identical to $\mathcal{H}(p, r)$. Thus, the lower bound carries over (with a smaller constant). By Lemma 5.3.3, $\mathcal{H}^*$ is pairwise independent with uniformly distributed function values. $\square$

We remark that the lower bound is tight. A corresponding $O(n \log n)$ upper bound can be shown by applying the framework of section 5.4, but using Chebychev's inequality rather than Lemma 5.2.1 as the basic tool for bounding probabilities.

5.4   5-wise independence

We want to bound the expected number of probes into the table made during any single operation (insertion, deletion, or lookup of a key x) when the hash table contains the set S of keys. It is well known that for linear probing, the set P of occupied table positions depends only on the set S and the hash function, independent of the sequence of insertions and deletions performed. An operation on key x makes no more than

$$1 + \max\{l \mid h(x) + [l] \subseteq P\}$$

probes into the table, because the iteration stops when the next unoccupied position is found (or sooner in case of a successful search). We first show a lemma which intuitively says that if the operation on the key x goes on for at least l steps, then there are either "many" keys hashing to the interval $h(x) + [l]$, or there are "many" keys that hash to some interval having $h(x)$ as its right endpoint.

Lemma 5.4.1. *For any $l > 0$ and $\bar{\alpha} \in (0,1)$, if $h(x) + [l] \subseteq P$ then at least one of the following holds:*

1. $\left|\{y \in S \setminus \{x\} \ : \ h(y) \in (h(x) + [l])\}\right| \geq \frac{1+\bar{\alpha}}{2}l - 1$, or

2. $(\exists \ell) \left|\{y \in S \ : \ h(y) \in [h(x) - \ell, h(x))\}\right| \geq \ell + \frac{1-\bar{\alpha}}{2}l$.

Proof. Suppose that $\left|\{y \in S \setminus \{x\} \ : \ h(y) \in (h(x) + [l])\}\right| < \frac{1+\bar{\alpha}}{2}l - 1$. Then in either case, $x \in S$ or $x \notin S$, it holds that $\left|\{y \in S \ : \ h(y) \in (h(x) + [l])\}\right| < \frac{1+\bar{\alpha}}{2}l$. Let

$$l' \stackrel{\text{def}}{=} \max\{\ell \ : \ [h(x) - \ell, h(x)] \subseteq P\} \ .$$

Now, fix any way of placing the keys in the hash table, e.g., suppose that keys are inserted in sorted order. Consider the set $S^* \subseteq S$ of keys stored in the interval $I = [h(x) - l', h(x) + l - 1]$. By the choice of l' there must be an empty position to the left of I, so $h(S^*) \subseteq I$. This means:

$$\begin{aligned} \left|\{y \in S \ : \ h(y) \in [h(x) - l', h(x))\}\right| &\geq \left|\{y \in S^* \ : \ h(y) \in [h(x) - l', h(x))\}\right| \\ &\geq |S^*| - \left|\{y \in S^* \ : \ h(y) \in (h(x) + [l])\}\right| \\ &> |I| - \frac{1+\bar{\alpha}}{2}l \\ &= l' + \frac{1-\bar{\alpha}}{2}l \ . \end{aligned}$$

$\square$

5.4.1   A simple bound

We start out with a bound that is simpler to derive than our final bound in section 5.4.2. It is possible to skip this section, but we believe that reading it makes it easier to understand the more complicated argument in section 5.4.2.

The next lemma upper bounds the probability that there exists some interval of form $[h(x) - \ell, h(x))$ having $\ell + d$ keys hashing into it, with d being a parameter. The derived bound will later be used to cover the case 2 from Lemma 5.4.1.

Lemma 5.4.2. *Let $S \subseteq U$ be a set of size n, and $\mathcal{H}$ a 5-wise independent family of functions from U to R with a maximum load of $\bar{\alpha} < 1$. If h is chosen uniformly at random from $\mathcal{H}$, then for any $x \in U$ and $\lambda > 0$,*

$$\Pr\left\{\max_\ell \left(\left|\{y \in S \ : \ h(y) \in [h(x) - \ell, h(x))\}\right| - \ell\right) \geq \frac{\lambda + 1}{(1 - \bar{\alpha})^{3/2}}\right\} < \frac{8\bar{\alpha}}{\lambda^2} \ .$$

Proof. We will use the symbol Δ to denote $\lceil\frac{\lambda}{2}(1 - \bar{\alpha})^{-3/2}\rceil$. Let A_i be the event that

$$\left|\{y \in S \ : \ h(y) \in [h(x) - i\Delta, h(x))\}\right| - i\Delta \geq \Delta \ .$$

We claim that it is sufficient to show $\Pr\left(\bigcup_{i>0} A_i\right) < \frac{8\bar{\alpha}}{\lambda^2}$. To see this, suppose that $\left|\{y \in S \ : \ h(y) \in [h(x) - \ell, h(x))\}\right| - \ell \geq \frac{\lambda + 1}{(1 - \bar{\alpha})^{3/2}}$, for some ℓ. Let $i' = \lceil\frac{\ell}{\Delta}\rceil$. Then

$$
\begin{aligned}
\left|\{y \in S \ : \ h(y) \in [h(x) - i'\Delta, h(x))\}\right| &\geq \ell + \frac{\lambda + 1}{(1 - \bar{\alpha})^{3/2}} \\
&\geq i'\Delta - (\Delta - 1) + \lambda(1 - \bar{\alpha})^{-3/2} + 1 \\
&\geq i'\Delta + \frac{\lambda}{2}(1 - \bar{\alpha})^{-3/2} + 1 \geq i'\Delta + \Delta \ .
\end{aligned}
$$

In this lemma we use a simple upper bound $\Pr(\bigcup_{i>0} A_i) \leq \sum_{i>0} \Pr(A_i)$. We use Lemma 5.2.1 to estimate each value $\Pr(A_i)$. Note that intersections of any interval $[h(x) - \ell, h(x))$ with the sets $h(S \setminus \{x\})$ and $h(S)$ are the same.

$$\sum_{i>0} \Pr(A_i) \leq \sum_{i>0} \frac{4\bar{\alpha}(i\Delta)^2}{((1 - \bar{\alpha})i\Delta + \Delta)^4} \leq \frac{4\bar{\alpha}}{\Delta^2} \sum_t \frac{(\frac{t}{1-\bar{\alpha}})^2}{(t + 1)^4}$$

We used the substitution $t = (1 - \bar{\alpha})i$. The last sum is over $t \in \{1 - \bar{\alpha}, \ 2(1 - \bar{\alpha}), \ldots\}$. The function $\frac{t^2}{(1+t)^4}$ is first increasing and then decreasing on $[0, \infty)$. Thus the sum can be bounded by the integral $\frac{1}{1-\bar{\alpha}} \int_{1-\bar{\alpha}}^\infty \frac{t^2}{(1+t)^4}dt$ plus the value of the biggest term in the sum.

$$
\begin{aligned}
\sum_{i>0} \Pr(A_i) &< \frac{4\bar{\alpha}}{\Delta^2} \frac{1}{(1 - \bar{\alpha})^2} \left(\max_{t>0} \frac{t^2}{(1 + t)^4} + \frac{1}{1 - \bar{\alpha}} \int_{1-\bar{\alpha}}^\infty \frac{t^2}{(1 + t)^4}dt\right) \\
&< \frac{4\bar{\alpha}}{\Delta^2} \frac{1}{(1 - \bar{\alpha})^3} \left(\frac{1}{10} + \int_0^\infty \frac{t^2}{(1 + t)^4}dt\right) \\
&< \frac{2\bar{\alpha}}{\Delta^2}(1 - \bar{\alpha})^{-3} \leq \frac{8\bar{\alpha}}{\lambda^2} \ .
\end{aligned}
$$

$\square$

Theorem 5.4.3. *Consider any sequence of operations (insertions, deletions, and lookups) in a linear probing hash table where the hash function h used has been chosen uniformly at random from a 5-wise independent family of functions $\mathcal{H}$. Let n and $\bar{\alpha} < 1$ denote, respectively, the maximum number of keys in the table during a particular operation and the corresponding maximum load. Then the expected number of probes made during that operation is $O((1 - \bar{\alpha})^{-5/2})$.*

Proof. We refer to x, S, and P as defined previously in this section. As argued above, the expected probe count is bounded by

$$1 + \sum_{l>0} \Pr\{h(x) + [l] \subseteq P\} \ .$$

Let $l_0 = \frac{10}{(1-\bar{\alpha})^{5/2}}$. For $l \leq l_0$ we use the trivial upper bound $\Pr\{h(x) + [l] \subseteq P\} \leq 1$. In the following we consider the case $l > l_0$.

Let A_l be the event that $\left|\{y \in S \setminus \{x\} \ : \ h(y) \in (h(x) + [l])\}\right| \geq \frac{1+\bar{\alpha}}{2}l - 1$, and let B_l be the event that $(\exists \ell) \left|\{y \in S \ : \ h(y) \in [h(x) - \ell, \ h(x))\}\right| \geq \ell + \frac{1-\bar{\alpha}}{2}l$. Lemma 5.4.1 implies that

$$\sum_{l>l_0} \Pr\{h(x) + [l] \subseteq P\} \leq \sum_{l>l_0} \big(\Pr(A_l) + \Pr(B_l)\big) \ .$$

Estimates of $\Pr(A_l)$ and $\Pr(B_l)$ are obtained from Lemma 5.2.1 and Lemma 5.4.2 respectively:

$$
\begin{aligned}
\sum_{l>l_0} \big(\Pr(A_l) + \Pr(B_l)\big) \ &< \ \sum_{l>l_0} \left(\frac{4\bar{\alpha}l^2}{(\frac{1-\bar{\alpha}}{2}l - 1)^4} + \frac{8\bar{\alpha}}{(\frac{(1-\bar{\alpha})^{5/2}}{2}l - 1)^2} \right) \\
&= \ O\left(\bar{\alpha} \sum_{l>l_0} \big((1-\bar{\alpha})^{-4}l^{-2} + (1-\bar{\alpha})^{-5}l^{-2}\big) \right) \\
&= \ O\left((1-\bar{\alpha})^{-5}/l_0\right) = O((1-\bar{\alpha})^{-5/2}) \ .
\end{aligned}
$$

$\square$

5.4.2 Improving the bound

By inspecting the proof of Theorem 5.4.3, one notices that an improvement to the result of Lemma 5.4.2 directly leads to an improvement of the main upper bound. The following lemma gives a bound with better dependence on α, which is significant for high load factors. The stated constant factor is far from being tight. Showing a considerably better constant factor would require a more tedious proof with inelegant calculations.

Lemma 5.4.4. *Let $S \subseteq U$ be a set of size n, and $\mathcal{H}$ a 5-wise independent family of functions from U to R with a maximum load of $\bar{\alpha}$. If h is chosen uniformly at random from $\mathcal{H}$, then for any $x \in U$,*

$$\Pr\left\{ \max_{\ell} \left(\left|\{y \in S \ : \ h(y) \in [h(x) - \ell, \ h(x))\}\right| - \ell\right) \geq \frac{\lambda + 2}{(1-\bar{\alpha})^{7/6}} \right\} < \frac{500\bar{\alpha}}{\lambda^2} \ .$$

Proof. We will use the symbol Δ to denote $\lceil \frac{\lambda}{3}(1-\bar{\alpha})^{-7/6}\rceil$. Let A_i' be the event that

$$\left|\{y \in S \ : \ h(y) \in [h(x) - i\Delta, \ h(x))\}\right| - i\Delta \geq 2\Delta \ .$$

It is sufficient to find a good upper bound on $\Pr(\bigcup_{i>0} A_i')$. To see this, suppose that $\left|\{y \in S \ : \ h(y) \in [h(x) - \ell, \ h(x))\}\right| - \ell \geq \frac{\lambda+2}{(1-\bar{\alpha})^{7/6}}$, for some ℓ. Let $i' = \lceil \frac{\ell}{\Delta} \rceil$.

Then

$$
\begin{aligned}
\left| \{ y \in S \ : \ h(y) \in [h(x) - i'\Delta, \ h(x)) \} \right| \ &\geq \ \ell + \frac{\lambda + 2}{(1 - \bar{\alpha})^{7/6}} \\
&\geq \ i'\Delta - (\Delta - 1) + \lambda(1 - \bar{\alpha})^{-7/6} + 2 \\
&\geq \ i'\Delta + 2\frac{\lambda}{3}(1 - \bar{\alpha})^{-7/6} + 2 \geq i'\Delta + 2\Delta \ .
\end{aligned}
$$

We define the events A_i by $A_i = A_i' \setminus \bigcup_{j > i} A_j'$. It holds that $A_i \cap A_j = \emptyset$, $i \neq j$, and $\bigcup_{i > 0} A_i = \bigcup_{i > 0} A_i'$. Therefore, $\Pr(\bigcup_{i > 0} A_i') = \sum_{i > 0} \Pr(A_i)$.

For the purpose of determining certain constraints that values $\Pr(A_i)$ must satisfy, we define the events B_i and C_{ij} by

$$
B_i = \left\{ h \in \mathcal{H} \ : \ \left| \{ y \in S \ : \ h(y) \in [h(x) - i\Delta, \ h(x)) \} \right| \geq \frac{1 + \bar{\alpha}}{2} i\Delta + \Delta \right\}
$$

and

$$
C_{ij} = \left\{ h \in \mathcal{H} \ : \ \left| \{ y \in S \ : \ h(y) \in [h(x) - i\Delta, \ h(x) - j\Delta \} \right| \geq \frac{1 - \bar{\alpha}}{2} j\Delta + (i - j + 1)\Delta \right\}
$$

for $i > j > 0$. Intuitively, B_i is the event that A_i' *nearly* holds, with no more than $(1 + \frac{1 - \bar{\alpha}}{2} i)\Delta$ elements missing in the interval. C_{ij} is the event that the interval $[h(x) - i\Delta, \ h(x) - j\Delta)$ contains the hash values of at least $\frac{1 - \bar{\alpha}}{2} j\Delta$ more elements than the size of the interval. For $k < i$, it holds that

$$
A_i \subseteq A_i' \subseteq B_{i-k} \cup C_{i, i-k} \ . \tag{5.3}
$$

Hence, for a fixed j,

$$
\bigcup_{i > j} (A_i \setminus C_{ij}) \subset B_j \ ,
$$

and as a result $\sum_{i > j} \Pr(A_i \setminus C_{ij}) \leq \Pr(B_j)$. Summing over $j > 0$ and re-expressing the sums we get:

$$
\sum_{i > 0} \sum_{j > 0} \Pr(A_i \setminus C_{ij}) \leq \sum_{i > 0} \Pr(B_i) \ . \tag{5.4}
$$

We will first estimate $\sum_i \Pr(B_i)$ using Lemma 5.2.1 (note that intersections of any interval $[h(x) - \ell, \ h(x))$ with the sets $h(S \setminus \{x\})$ and $h(S)$ are the same):

$$
\begin{aligned}
\sum_{i > 0} \Pr(B_i) \ &\leq \ \sum_{i > 0} \frac{4\bar{\alpha}(i\Delta)^2}{(\frac{1 - \bar{\alpha}}{2} i\Delta + \Delta)^4} \\
&< \ \frac{4\bar{\alpha}}{\Delta^2} \frac{8}{(1 - \bar{\alpha})^3} \left(\max_{t > 0} \frac{t^2}{(1 + t)^4} + \int_{1 - \bar{\alpha}}^{\infty} \frac{t^2}{(1 + t)^4} dt \right) \\
&< \ \frac{4\bar{\alpha}}{\Delta^2} \frac{8}{(1 - \bar{\alpha})^3} \left(\frac{1}{10} + \int_0^{\infty} \frac{t^2}{(1 + t)^4} dt \right) \\
&< \ \frac{16\bar{\alpha}}{\Delta^2} (1 - \bar{\alpha})^{-3} \ .
\end{aligned}
$$

Again using Lemma 5.2.1 to estimate $\Pr(C_{ij})$, for $i > j > 0$, we get:

$$
\Pr(C_{ij}) \leq \frac{4\bar{\alpha}(i - j)^2 \Delta^2}{(\frac{1 - \bar{\alpha}}{2}(2i - j)\Delta + \Delta)^4} < \frac{4\bar{\alpha}}{\Delta^2} \frac{(i - j)^2}{(\frac{1 - \bar{\alpha}}{2} i + 1)^4} \ .
$$

For the probability of the event $A_i \setminus C_{ij}$ we use a trivial lower bound of $\Pr(A_i) - \Pr(C_{ij})$. Hence, for any $i > 0$,

$$\sum_{k=1}^{i-1} \Pr(A_i \setminus C_{i,i-k}) > \sum_{k=1}^{i-1} \max \left\{ 0, \ \Pr(A_i) - \frac{4\bar{\alpha}}{\Delta^2} \frac{k^2}{(\frac{1-\bar{\alpha}}{2}i + 1)^4} \right\} \ .$$

Let $p_i = \Pr(A_i)$, $\gamma_i = \frac{4\bar{\alpha}}{\Delta^2}(\frac{1-\bar{\alpha}}{2}i + 1)^{-4}$, and $k_i = \lfloor \sqrt{p_i/\gamma_i} \rfloor$. Lemma 5.2.1 gives $p_i < \frac{4\bar{\alpha}(i\Delta)^2}{((1-\bar{\alpha})i\Delta + 2\Delta)^4}$, and so $k_i \leq \lfloor i/4 \rfloor \leq i - 1$. We further have that

$$k_i p_i - \gamma_i \sum_{k=1}^{k_i} k^2 > p_i(\sqrt{p_i/\gamma_i} - 1) - \gamma_i \left(\frac{(p_i/\gamma_i)^{3/2}}{2} + 1 \right) = \frac{p_i^{3/2}}{2\sqrt{\gamma_i}} - p_i - \gamma_i \ .$$

An upper bound on $\sum_{i>0} p_i$ can be obtained through solving an optimization problem over real variables x_1, x_2, ..., $x_{\lfloor r/\Delta \rfloor}$. The problem is to maximize $\sum_{i>0} x_i$ subject to the constraint that

$$\sum_{i>0} \frac{x_i^{3/2}}{2\sqrt{\gamma_i}} - x_i \leq \frac{16\bar{\alpha}}{\Delta^2}(1 - \bar{\alpha})^{-3} + \sum_{i>0} \gamma_i \ .$$

The vector $(p_1, p_2, \ldots)$ is a feasible solution to the problem and thus $\sum_{i>0} p_i$ is not larger than the optimal value. By employing the method of Lagrange multipliers we find that the optimal solution is $x_i = \gamma_i y^2$, where y is a value that satisfies:

$$y^3 - y^2 = \frac{4}{(1 - \bar{\alpha})^3} \frac{1}{\sum_{i>0}(\frac{1-\bar{\alpha}}{2}i + 1)^{-4}} + 1 \ .$$

We have that $\frac{2}{3(1-\bar{\alpha})} < \sum_{i>0}(\frac{1-\bar{\alpha}}{2}i+1)^{-4} < 1 + \frac{2}{3(1-\bar{\alpha})}$. Suppose that $y > 2.5$, so we may write $\frac{1}{2}y^3 < y^3 - y^2 - 1$. It follows that $y < (\frac{16}{(1-\bar{\alpha})^2})^{1/3}$. Since $\sqrt[3]{16} > 2.5$ the calculated upper bound on y is true in general. Finally,

$$\sum_{i>0} p_i < \sum_{i>0} \gamma_i y^2 \ < \ \left(\frac{16}{(1 - \bar{\alpha})^2} \right)^{2/3} \frac{72\bar{\alpha}}{\lambda^2}(1 - \bar{\alpha})^{14/6-1} < \frac{500\bar{\alpha}}{\lambda^2} \ .$$

$\square$

By changing l_0 to $\frac{50}{(1-\bar{\alpha})^{13/6}}$ in the proof of Theorem 5.4.3, and utilizing the previous lemma, the following result is proved.

Theorem 5.4.5. *Consider any sequence of operations (insertions, deletions, and lookups) in a linear probing hash table where the hash function h used has been chosen uniformly at random from a 5-wise independent family of functions $\mathcal{H}$. Let n and $\bar{\alpha} < 1$ denote, respectively, the maximum number of keys in the table during a particular operation and the corresponding maximum load. Then the expected number of probes made during that operation is $O((1-\bar{\alpha})^{-13/6})$.*

5.5 Blocked probing

In this section we propose and analyze a family of open addressing methods, containing among other a variant of bidirectional linear probing. The expected probe count for any single operation is within a constant factor from the corresponding value in linear probing with a fully random hash function. For successful searches we do a more precise analysis. It is shown that the expected number of probes made during a search for a random element of S is less than $1 + \frac{2}{1-\bar{\alpha}} \frac{\bar{\alpha}}{\alpha}$. The bounds for single operations require a 5-wise independent family of functions. The bound for average successful search requires 4-wise independence.

Suppose that keys are hashed into a table of size r by a function h. For simplicity we assume that r is a power of two. Define $x \ominus a = x - (x \bmod a)$, for any integers x and a. Let $V_j^i = \{j, j+1, \ldots, j + 2^i - 1\}$, where j is assumed to be a multiple of 2^i. The intervals V_j^i may be thought of as sets of references to slots in the hash table. In a search for key x, intervals V_j^i that enclose $h(x)$ are examined in the order of increasing i. More precisely, $V_{h(x)}^0$ is examined first; if the search did not finish after traversing $V_{h(x) \ominus 2^i}^i$, then the search proceeds in the untraversed half of $V_{h(x) \ominus 2^{i+1}}^{i+1}$. The search stops after traversal of an interval if any of the following three cases hold:

  a) key x was found,

  b) the interval contained empty slot(s),

  c) the interval contained key(s) whose hash value does not belong to the interval.

In case (a) the search may obviously stop immediately on discovery of x — there is no need to traverse through the rest of the interval. In cases (b) and (c) we will be able to conclude that x is not in the hash table.

Traversal of unexamined halves of intervals V_j^i may take different concrete forms; the only requirement is that every slot is probed exactly once. From a practical point of view, a good choice is to probe slots sequentially in a way that makes the scheme a variant of bidirectional linear probing. This concrete version defines a probe sequence that in probe numbers 2^i to $2^{i+1} - 1$ inspects either slots

$$(h(x) \ominus 2^i + 2^i, \ h(x) \ominus 2^i + 2^i + 1, \ldots, \ h(x) \ominus 2^i + 2^{i+1} - 1)$$

$$\text{or} \quad (h(x) \ominus 2^i - 1, \ h(x) \ominus 2^i - 2, \ldots, \ h(x) \ominus 2^i - 2^i)$$

depending on whether $h(x) \bmod 2^i = h(x) \bmod 2^{i+1}$ or not. A different probe sequence that falls in this class of methods, but is not sequential, is $(x, j) \mapsto h(x) \oplus j$, with j starting from 0.

Insertions. Until key x which is being inserted is placed in a slot, the same probe sequence is followed as in a search for x. However, x may be placed in a non-empty slot if its hash value is closer to the slot number in a special metric which we will now define. Let $d(y_1, y_2) = \min\{i \mid y_2 \in V_{y_1 \ominus 2^i}^i\}$. The value of

$d(y_1, y_2)$ is equal to the position of the most significant bit in which y_1 and y_2 differ. If during insertion of x we encounter a slot y containing key x' such that $d(h(x), y) < d(h(x'), y)$ then key x is put into slot y. In an implementation there is no need to evaluate $d(h(x), y)$ values every time. We can keep track of what interval $V^i_{h(x)\ominus 2^i}$ is being traversed at the moment and check whether $h(x')$ belongs to that interval.

When x is placed in slot y which was previously occupied by x', a new slot for x' has to be found. Let $i = d(h(x'), y)$. The procedure now continues as if x' is being inserted and we are starting with traversal of $V^i_{h(x')\ominus 2^i} \setminus V^{i-1}_{h(x')\ominus 2^{i-1}}$. If the variant of bidirectional linear probing is used, the traversal may start from position y, which may matter in practice.

Deletions. After removal of a key we have to check if the new empty slot can be used to bring some keys closer to their hash values, in terms of the metric d. Let x be the removed key, y be the slot in which it resided, and $i = d(h(x), y)$. There is no need to examine $V^{i-1}_{h(x)\ominus 2^{i-1}}$. If $V^i_{h(x)\ominus 2^i} \setminus V^{i-1}_{h(x)\ominus 2^{i-1}}$ contains another empty slot then the procedure does not continue in wider intervals. If it continues and an element gets repositioned then the procedure is recursively applied starting from the new empty slot.

It is easy to formally check that appropriate invariants hold and that the above described set of procedures works correctly.

5.5.1 Analysis

We analyze the performance of operations on a hash table of size r when blocked probing is used. Suppose that the hash table stores an arbitrary fixed set of n elements, and let $\bar{\alpha}$ denote the maximum load of $\mathcal{H}$ on sets of size n. Let $C^U_{\bar{\alpha}}$, $C^I_{\bar{\alpha}}$, $C^D_{\bar{\alpha}}$, and $C^S_{\bar{\alpha}}$ be the random variables that represent, respectively: the number of probes made during an unsuccessful search for a fixed key, the number of probes made during an insertion of a fixed key, the number of probes made during a deletion of a fixed key, and the number of probes made during a successful search for a random element from the set. In the above notation we did not explicitly include the fixed set and fixed elements that are used in the operations; they have to be implied by the context. The upper bounds on the expectations of $C^\Xi_{\bar{\alpha}}$ variables, which are given by the following theorem, do not depend on choices of those elements.

In this section, upper bounds have explicit constant factors. We introduce a function that appears in statements of some upper bounds. Define $T(\bar{\alpha})$ as the function over domain $(0, 1)$ with the value given by

$$T(\bar{\alpha}) = \begin{cases} \frac{5.2\bar{\alpha}}{(1-\bar{\alpha})^2} + \frac{1}{3} & \text{for } \bar{\alpha} \geq \frac{1}{3}, \\ \frac{2.5\bar{\alpha}}{(1-\bar{\alpha})^4} & \text{for } \bar{\alpha} < \frac{1}{3}. \end{cases}$$

It can be checked that $T(\bar{\alpha}) < \frac{5.7\bar{\alpha}}{(1-\bar{\alpha})^2}$. By $\mathrm{frac}(x)$ we denote the function $x \mapsto x - \lfloor x \rfloor$.

Theorem 5.5.1. *Let $\mathcal{H}$ be a 5-wise independent family of functions which map U to R. For a maximum load of $\bar{\alpha} < 1$, blocked probing with a hash function*

chosen uniformly at random from $\mathcal{H}$ provides the following expectations.

$$
\begin{aligned}
E(C_{\bar{\alpha}}^U) &< 1 + T(\bar{\alpha}) \\
E(C_{\bar{\alpha}}^I) &< 1 + 2T(\bar{\alpha}) \\
E(C_{\bar{\alpha}}^D) &< 1 + 2T(\bar{\alpha})
\end{aligned}
$$

Proof. Denote by x the fixed element from $U \setminus S$ that is being inserted or searched (unsuccessfully). Let $\bar{C}_{\bar{\alpha}}^U$ be the random variable that takes value 2^i when $2^{i-1} < C_{\bar{\alpha}}^U \leq 2^i$, $0 \leq i \leq \lg r$. We can write $\bar{C}_{\bar{\alpha}}^U = 1 + \sum_{i=1}^{\lg r} 2^{i-1} T_i$, where T_i is an indicator variable whose value is 1 when at least $2^{i-1} + 1$ probes are made during the search. Let A_j be the event that the interval of slots $V_{h(x) \ominus 2^j}^j$ is *fully loaded*, meaning that at least 2^j elements of S are hashed into the interval. If $T_i = 1$ then A_{i-1} holds, so we have:

$$
E(\bar{C}_{\bar{\alpha}}^U) \leq 1 + \sum_{i=1}^{\lg r} 2^{i-1} \Pr(A_{i-1}) \ .
$$

An upper bound on $\Pr(A_i)$ can be derived from Lemma 5.2.1. Denoting $K = \frac{3\bar{\alpha}^2}{(1-\bar{\alpha})^4}$, we may write $\Pr(A_i) < \frac{K}{2^{2j}} + \frac{K}{3\bar{\alpha} \cdot 2^{3j}}$ (we used the bound from the end of the proof of the lemma; that bound was simplified in the lemma statement). Yet, for small lengths, and $\bar{\alpha}$ not small, the aforementioned bound is useless; then we will simply use the trivial upper bound of 1. We first consider the case $K \geq 1$. Denoting $j_* = \lceil \frac{1}{2} \lg K \rceil$ we have that

$$
\begin{aligned}
\mathrm{E}(C_{\bar{\alpha}}^U - 1) &\leq 2^{j_*} - 1 + \sum_{j=j_*}^{\lg r} 2^j \left(\frac{K}{2^{2j}} + \frac{K}{3\bar{\alpha} \cdot 2^{3j}} \right) \\
&= 2^{j_*} - 1 + K \sum_{j=j_*}^{\lg r} 2^{-j} + \frac{K}{3\bar{\alpha}} \sum_{j=j_*}^{\lg r} 2^{-2j} \\
&< 2^{j_*} - 1 + \frac{K}{2^{j_*}} \frac{1}{1 - \frac{1}{2}} + \frac{K}{3\bar{\alpha} \cdot 4^{j_*}} \frac{1}{1 - \frac{1}{4}} \\
&\leq \sqrt{K} \cdot 2^{1 - \mathrm{frac}(\lg \sqrt{K})} - 1 + 2\sqrt{K} \cdot 2^{-(1 - \mathrm{frac}(\lg \sqrt{K}))} + \frac{4}{9\bar{\alpha}} \\
&\leq 3\sqrt{K} + \frac{4}{9\bar{\alpha}} - 1 \ .
\end{aligned}
$$

The last inequality is true because $2^t + 2 \cdot 2^{-t} \leq 3$, for $t \in [0,1]$. The assumption that $K \geq 1$ implies that $\bar{\alpha} > 0.29$, but we decide to use the obtained bound for $\bar{\alpha} \geq \frac{1}{3}$. Thus, we may replace $\frac{4}{9\bar{\alpha}} - 1$ with an upper bound of $\frac{1}{3}$. Doing an easier calculation without splitting of the sum at index j_* gives $\mathrm{E}(C_{\bar{\alpha}}^U) < 1 + K(2 + \frac{4}{9\bar{\alpha}})$. When $\bar{\alpha} < \frac{1}{3}$, we may replace $K(2 + \frac{4}{9\bar{\alpha}})$ with an upper bound of $\frac{2.5\bar{\alpha}}{(1-\bar{\alpha})^4}$.

We now move on to analyzing insertions. Let $\bar{C}_{\bar{\alpha}}^I$ be the random variable that takes value 2^i when $2^{i-1} < C_{\bar{\alpha}}^I \leq 2^i$, $0 \leq i \leq \lg r$. The variable $C_{\bar{\alpha}}^U$ gives us the slot where x is placed, but we have to consider possible movements of other elements. If x is placed into a slot previously occupied by key x' from a "neighbouring" interval $V_{h(x) \ominus 2^{i+1}}^{i+1} \setminus V_{h(x) \ominus 2^i}^i$, then as many as 2^i probes may be necessary to find a place for x' in $V_{h(x) \ominus 2^i}^i$, if there is one. If $V_{h(x) \ominus 2^{i+1}}^{i+1}$ is fully

loaded, then as many as 2^{i+1} additional probes may be needed to find a place within $V_{h(x)\ominus2^{i+2}}^{i+2} \setminus V_{h(x)\ominus2^{i+1}}^{i+1}$, and so on. In general — and taking into account all repositioned elements — we use the following accounting to get an overestimate of $E(\bar{C}_{\bar{\alpha}}^I)$: For every fully loaded interval $V_{h(x)\ominus2^i}^i$ we charge 2^i probes, and for every fully loaded neighbouring interval $V_{h(x)\ominus2^{i+1}}^{i+1} \setminus V_{h(x)\ominus2^i}^i$ we also charge 2^i probes. The probability of a neighbouring interval of length 2^i being full is equal to $\Pr(A_i)$. As a result,

$$E(\bar{C}_{\bar{\alpha}}^I) \leq 1 + \sum_{i=0}^{\lg r - 1} 2^i \cdot 2\Pr(A_i) < 1 + 2T(\bar{\alpha}) \ .$$

The analysis of deletions is analogous. $\qquad\square$

For higher values of $\bar{\alpha}$, the dominant term in the upper bounds is $O(\frac{1}{(1-\bar{\alpha})^2})$. The constant factors in front of term $(1-\bar{\alpha})^{-2}$ are relatively high compared to standard linear probing with fully random hash functions. This is in part due to approximative nature of the proof of Theorem 5.5.1, and in part due to tail bounds that we use, which are weaker than those for fully independent families. In the fully independent case, the probability that an interval of length q is fully loaded is less than $e^{q(1-\alpha+\ln\alpha)}$, according to Chernoff-Hoeffding bounds [Che52, Hoe63]. Plugging this bound into the proof of Theorem 5.5.1 would give, for example,

$$E(C_\alpha^U) < 1 + \frac{e^{1-\alpha+\ln\alpha}}{\ln 2 \cdot |1 - \alpha + \ln\alpha|} \ . \tag{5.5}$$

For α close to 1, a good upper bound on (5.5) is $1 + \frac{2}{\ln 2}(1-\alpha)^{-2}$. The constant factor here is ≈ 2.88, as opposed to ≈ 5.2 from the statement of Theorem 5.5.1. As α gets smaller, the bound in (5.5) gets further below $1 + \frac{2}{\ln 2}(1-\alpha)^{-2}$.

5.5.2 Analysis of successful searches

As before, we assume that the function h is chosen uniformly at random from $\mathcal{H}$. For a subset Q of R, let X_i be the indicator random variable that has value 1 iff $h(x_i) \in Q$, $1 \leq i \leq n$. The variable $X = \sum_{i=1}^n X_i$ counts the number of elements that are mapped to Q. We introduce a random variable that counts the number of elements that have *overflowed* on Q. Define $Y = \max\{X - q, 0\}$, where $q = |Q|$. We will find an upper bound on $E(Y)$ that is expressed only in terms of q and $\bar{\alpha}_{\mathcal{H}}$. Denote such a bound by $M_q^{\bar{\alpha}}$. It is clear that

$$E(C_{\bar{\alpha}}^S) \leq 1 + \frac{1}{n}\sum_{l=0}^{\lg r - 1} 2^l \frac{r}{2^l} M_{2^l}^{\bar{\alpha}} = 1 + \frac{1}{\alpha}\sum_{l=0}^{\lg r - 1} M_{2^l}^{\bar{\alpha}} \ .$$

We are starting the analysis of $E(Y)$, for an arbitrary fixed set Q. The value of $E(Y)$ is $\sum_{j=1}^{n-q} j \cdot \Pr\{X = q + j\}$. A bound on $E(Y)$ can be obtained as the optimal value of an optimization problem, which we will introduce. We denote $E(X)$ shortly by μ. Let variables p_i, $0 \leq i \leq n$ have domain $[0,1]$. Define the

following optimization problem in variables $p_0, \ldots, p_n$:

$$\text{maximize} \qquad \sum_{j=1}^{n-q} j \cdot p_{q+j}$$

subject to constraints:

$$\sum_{i=0}^{n} p_i = 1 \, ,$$

$$\sum_{i=0}^{\lceil \mu \rceil - 1} (\mu - i) p_i = \sum_{i=\lfloor \mu \rfloor + 1}^{n} (i - \mu) p_i \, ,$$

$$\sum_{i=0}^{n} (\mu - i)^k p_i = D_k \, .$$

If we choose an even $k \geq 2$, and set D_k to be an upper bound on the value of the kth central moment of X, then the optimal value of the objective function is an upper bound on $E(Y)$. Remark that in an optimal solution $p_i = 0$ for $\lceil \mu \rceil \leq i \leq q$. We will actually not solve the above problem, but its relaxation. We introduce variables d_i, $i \in I = \{-\lceil \mu \rceil, \ldots, -2, -1, 1, 2, \ldots n - q\}$. For $i \in \{-\lceil \mu \rceil, \ldots, -1\}$ the domain of variables d_i is $(0, \mu]$; for $i \in \{1, \ldots n - q\}$ the domain of variables d_i is $(q - \mu, n]$. The variables representing probabilities are still denoted by p_i, but the index set is now I. Define optimization problem Π over all variables p_i, d_i as:

$$\text{maximize} \qquad \sum_{j=1}^{n-q} p_j (d_j - (q - \mu))$$

subject to constraints:

$$\sum_{i \in I} p_i = 1 \, , \tag{5.6}$$

$$\sum_{i=-1}^{\lceil \mu \rceil} d_i p_i = \sum_{i=1}^{n-q} d_i p_i \, , \tag{5.7}$$

$$\sum_{i \in I} d_i^k p_i = D_k \, . \tag{5.8}$$

The optimal value of the objective function for the problem Π is not smaller than the optimal value in the original problem.

Lemma 5.5.2. *Let $(\bar{p}, \bar{d})$ be an optimal solution to the problem Π such that $\bar{d}_i \neq \bar{d}_j$ for $0 < i < j$ and $0 > i > j$. Then only one value among $\bar{p}_1, \ldots, \bar{p}_{n-q}$ is not equal to 0, and only one value among $\bar{p}_{-1}, \ldots, \bar{p}_{-\lceil \mu \rceil}$ is not equal to 0.*

Proof. We will prove the claims for $i > 0$ and $i < 0$ separately. Suppose the contrary, that there are two non-zero values among $\bar{p}_1, \ldots, \bar{p}_{n-q}$. W.l.o.g. we may assume that $\bar{p}_1 > 0$ and $\bar{p}_2 > 0$. Define function $f(d_1, d_2)$ over $d_1, d_2 \geq 0$ as

$$f(d_1, d_2) = \bar{p}_1 (d_1 - (q - \mu)) + \bar{p}_2 (d_2 - (q - \mu)) \, .$$

We are interested in finding the maximum of f subject to constraint $\bar{p}_1 d_1^k + \bar{p}_2 d_2^k - D = 0$, where $D = \bar{p}_1 \bar{d}_1^k + \bar{p}_2 \bar{d}_2^k$. According to the method of Lagrange multipliers, the only point in the interior of the domain that is a candidate for an extreme point is $(\hat{d}, \hat{d})$, where $\hat{d} = \sqrt[k]{\frac{D}{\bar{p}_1 + \bar{p}_2}}$. We cannot establish the character of the point directly through an appropriate quadratic form, because the required forms evaluate to zero (the form should be positive definite or negative definite to establish the local minimum or maximum, respectively). However, by comparing $f(\hat{d}, \hat{d})$ to the values of f at boundary points $(\sqrt[k]{D/\bar{p}_1}, 0)$ and $(0, \sqrt[k]{D/\bar{p}_2})$, we find that the maximum is reached in the interior of the domain and it must be at point $(\hat{d}, \hat{d})$. It is not hard to show that $\hat{d} \in (q - \mu, n]$, by looking at the set determined by equation $\bar{p}_1 d_1^k + \bar{p}_2 d_2^k - D = 0$ and using the fact that $(\bar{d}_1, \bar{d}_2) \in (q - \mu, n] \times (q - \mu, n]$.

We construct a new solution $(\tilde{p}, \tilde{d})$ to the problem Π in two phases. In the first phase: set $\tilde{p}_i = \bar{p}_i$ and $\tilde{d}_i = \bar{d}_i$ for $i \notin \{1, 2\}$, set $\tilde{p}_2 = \tilde{d}_2 = 0$, $\tilde{d}_1 = \hat{d}$ and $\tilde{p}_1 = \frac{\bar{p}_1 \bar{d}_1 + \bar{p}_2 \bar{d}_2}{\hat{d}}$. From $(\hat{d}, \hat{d})$ being the point of maximum of f, it follows that

$$\frac{\bar{p}_1 \bar{d}_1 + \bar{p}_2 \bar{d}_2}{\hat{d}} < \bar{p}_1 + \bar{p}_2 \ .$$

Now $(\tilde{p}, \tilde{d})$ exactly satisfies (5.7) and it satisfies the inequality conditions corresponding to (5.6) and (5.8). Already now, the value of the objective function is higher than the value at $(\bar{p}, \bar{d})$. It increases even further when we increase values of $\tilde{p}_1$ and $\tilde{p}_{-1}$ in a way that satisfies all the conditions with equalities. We reached a contradiction, meaning that $(\bar{p}, \bar{d})$ cannot have two non-zero values among $\bar{p}_1, \ldots, \bar{p}_{n-q}$.

Suppose that there are two non-zero values among $\bar{p}_{-1}, \ldots, \bar{p}_{-\lceil \mu \rceil}$. W.l.o.g. we may assume that $\bar{p}_{-1} > 0$ and $\bar{p}_{-2} > 0$. To reach a contradiction we use an argument that is in some way dual to the argument for the first part. Define function $g(d_{-1}, d_{-2}) = \bar{p}_{-1} d_{-1}^k + \bar{p}_{-2} d_{-2}^k$. Now we are interested in finding the minimum of g subject to constraint $\bar{p}_{-1} d_{-1} + \bar{p}_{-2} d_{-2} - (\bar{p}_{-1} \bar{d}_{-1} + \bar{p}_{-2} \bar{d}_{-2}) = 0$. The proof continues similarly to the first part. $\qquad \square$

Lemma 5.5.3. *Let* OPT *be the optimal value of the objective function for the problem* Π. *Then* OPT $< \frac{1}{2} \sqrt[k]{D_k}$, *and also*

$$\text{OPT} < \begin{cases} \frac{1}{2}^{\frac{k-1}{k}} \sqrt[k]{D_k} - \frac{1}{2}(q - \mu) & \text{when } \frac{D_k(1 - \frac{1}{k})^k}{(q - \mu)^k} \geq \frac{1}{2}, \\ \frac{D_k}{(q - \mu)^{k-1}} ((1 - \frac{1}{k})^{k-1} - (1 - \frac{1}{k})^k) & \text{otherwise.} \end{cases}$$

Proof. According to Lemma 5.5.2, an optimal solution to Π can be obtained from the following simplified problem:

$$\text{maximize} \qquad p_1(d_1 - (q - \mu))$$
$$\text{subject to constraints:}$$
$$p_1 d_1 = (1 - p_1) d_{-1} \ ,$$
$$p_1 d_1^k + (1 - p_1) d_{-1}^k = D_k \ .$$

Eliminating d_{-1} yields the constraint $d_1^k(p_1 + \frac{p_1^k}{(1-p_1)^{k-1}}) = D_k$. Expressing d_1 from this constraint shows that the objective function is equivalent to

$$f(p_1) = \frac{\sqrt[k]{D_k}}{\sqrt[k]{p_1^{1-k} + (1-p_1)^{1-k}}} - p_1(q - \mu)$$

The value of the first term is symmetrical around the point $\frac{1}{2}$. Therefore, the maximum of f is reached in the interval $(0, \frac{1}{2}]$. By analyzing only the first term, we get a simple upper bound of $\frac{1}{2}\sqrt[k]{D_k}$. To get the other bound we look at the function

$$\bar{f}(p_1) = \sqrt[k]{D_k} \cdot p_1^{1-1/k} - p_1(q - \mu) \ ,$$

which satisfies $f < \bar{f}$. Analyzing $\bar{f}'$ we easily find the maximum of $\bar{f}(p_1)$ over $p_1 \in (0, \frac{1}{2}]$. $\qquad\square$

Corollary 5.5.4. *Suppose that $\mathcal{H}$ is 4-wise independent. Define*

$$M_q^{\bar{\alpha}} = \begin{cases} \frac{1}{2}\sqrt{\bar{\alpha}q} & \text{If } \frac{\bar{\alpha}}{(1-\bar{\alpha})^2 q} \geq (\sqrt{2}+1)^2 \ , \\ \frac{1}{\sqrt{2}}\sqrt{\bar{\alpha}q} - \frac{1}{2}q(1-\bar{\alpha}) & \text{If } 2 \leq \frac{\bar{\alpha}}{(1-\bar{\alpha})^2 q} < (\sqrt{2}+1)^2 \ , \\ \frac{1}{4}\frac{\bar{\alpha}}{1-\bar{\alpha}} & \text{If } \frac{1}{\sqrt{2}} \leq \frac{\bar{\alpha}}{(1-\bar{\alpha})^2 q} < 2 \ , \\ 0.11 \cdot \frac{3\bar{\alpha}^2 q + \bar{\alpha}}{(1-\bar{\alpha})^3 q^2} & \text{If } \frac{\bar{\alpha}}{(1-\bar{\alpha})^2 q} < \frac{1}{\sqrt{2}} \ . \end{cases}$$

It holds that $E(Y) < M_q^{\bar{\alpha}}$.

Proof. For the first three cases we used $k = 2$, and for the fourth case $k = 4$ was used. The constants in the problem Π were substituted as: $\mu = \bar{\alpha}q$, $D_2 = \bar{\alpha}q$, and $D_4 = 3\bar{\alpha}^2 q^2 + \bar{\alpha}q$. The bound on the fourth central moment is proved very similarly to the proof of of Lemma 5.2.1. The bound on the variance is easier to prove. $\qquad\square$

The analysis is finalized with the following theorem.

Theorem 5.5.5. *Let $\mathcal{H}$ be a 4-wise independent family of functions which map U to R. When blocked probing is used with a hash function chosen uniformly at random from $\mathcal{H}$, then*

$$E(C_{\bar{\alpha}}^S) < 1 + \frac{\bar{\alpha}}{\alpha} \cdot \begin{cases} \frac{2}{1-\bar{\alpha}} & 0.5 \leq \bar{\alpha} \ , \\ \frac{1.1}{1-\bar{\alpha}} & 0.3 < \bar{\alpha} < 0.5 \ , \\ \frac{0.85}{1-\bar{\alpha}} & \bar{\alpha} \leq 0.3 \ . \end{cases}$$

Proof. As stated at the beginning of this section, $E(C_{\bar{\alpha}}^S)$ is upper-bounded by

$$1 + \frac{1}{\alpha} \sum_{l=0}^{\lg r - 1} M_{2^l}^{\bar{\alpha}} \ .$$

Now that we have $M_q^{\bar{\alpha}}$ values, we are only left to carry out summation of $\sum_l M_{2^l}^{\bar{\alpha}}$. We split the sum into four parts. The splitting indexes are set as follows: $l_1 = \lfloor \lg \frac{\bar{\alpha}}{(\sqrt{2}+1)^2(1-\bar{\alpha})^2} \rfloor$, $l_2 = \lfloor \lg \frac{\bar{\alpha}}{2(1-\bar{\alpha})^2} \rfloor$, $l_3 = \lfloor \lg \frac{\sqrt{2}\bar{\alpha}}{(1-\bar{\alpha})^2} \rfloor$. Some of the indices may

be smaller than 0, in which case the sum is split into fewer parts. For tighter bounding, we will also need the following values: $t_1 = \text{frac}\big(\lg \frac{\bar{\alpha}}{(\sqrt{2}+1)^2(1-\bar{\alpha})^2}\big)$, $t_2 = \text{frac}\big(\lg \frac{\bar{\alpha}}{2(1-\bar{\alpha})^2}\big)$, and $t_3 = \text{frac}\big(\lg \frac{\sqrt{2}\bar{\alpha}}{(1-\bar{\alpha})^2}\big)$.

Suppose first that $l_1 \geq 0$. Simple calculations yield the following four inequalities.

$$
\sum_{l=0}^{l_1} M_{2^l}^{\bar{\alpha}} < \frac{1}{\sqrt{2}}\frac{\bar{\alpha}}{1-\bar{\alpha}}2^{-t_1/2}
$$

$$
\sum_{l=l_1+1}^{l_2} M_{2^l}^{\bar{\alpha}} \leq \frac{\bar{\alpha}}{1-\bar{\alpha}}\Big(\big(1+\tfrac{1}{\sqrt{2}}\big)2^{-t_2/2} - 2^{-t_1/2} - \tfrac{1}{2}2^{-t_2} + \tfrac{1}{(\sqrt{2}+1)^2}2^{-t_1}\Big)
$$

$$
\sum_{l=l_2+1}^{l_3} M_{2^l}^{\bar{\alpha}} \leq \frac{1}{4}\frac{\bar{\alpha}}{1-\bar{\alpha}}(l_3 - l_2)
$$

$$
\sum_{l=l_3+1}^{\infty} M_{2^l}^{\bar{\alpha}} \leq 0.24\frac{\bar{\alpha}}{1-\bar{\alpha}}2^{t_3} + 0.02\frac{1-\bar{\alpha}}{\bar{\alpha}}2^{2t_3}
$$

For the third inequality, we notice that $l_3 - l_2 = 2$ for $t_3 \leq \frac{1}{2}$, and $l_3 - l_2 = 1$ for $t_3 > \frac{1}{2}$. The assumption $l_1 \geq 0$ implies that $\bar{\alpha} > \frac{1}{2}$ (we do not need the exact bound on $\bar{\alpha}$), which we use for the second term of the r.h.s. of the fourth inequality. Maximizing over t_1, t_2, t_3 shows that $\sum_{l=0}^{\infty} M_{2^l}^{\bar{\alpha}} < \frac{2\bar{\alpha}}{1-\bar{\alpha}}$.

Now suppose that $l_1 < 0$ and $l_2 \geq 0$. The first part of the sum is now

$$
\sum_{l=0}^{l_2} M_{2^l}^{\bar{\alpha}} \leq \frac{\bar{\alpha}}{1-\bar{\alpha}}\big(\big(1+\tfrac{1}{\sqrt{2}}\big)2^{-t_2/2} - \tfrac{1}{2}2^{-t_2}\big) - \big(1+\tfrac{1}{\sqrt{2}}\big)\sqrt{\bar{\alpha}} + \frac{1-\bar{\alpha}}{2} ,
$$

while the bounds on the other two parts stay the same. Since $l_1 < 0$, it follows that $\bar{\alpha} < 0.7$. When $\bar{\alpha} < 0.7$, it holds that $3\sqrt{\bar{\alpha}} > \frac{\bar{\alpha}}{1-\bar{\alpha}}$. On the other hand, $l_2 \geq 0$ is equivalent to $\bar{\alpha} \geq \frac{1}{2}$. When $\bar{\alpha} \geq \frac{1}{2}$, it holds that $\sqrt{\bar{\alpha}} > 1 - \bar{\alpha}$. Simple calculations again show that $\sum_{l=0}^{\infty} M_{2^l}^{\bar{\alpha}} < \frac{2\bar{\alpha}}{1-\bar{\alpha}}$ (a slightly lower constant could also be stated).

Now suppose that $l_2 < 0$ and $l_3 \geq 0$. This assumption implies that $\bar{\alpha} > 0.3$, and we may write $\frac{1-\bar{\alpha}}{\bar{\alpha}} < \frac{6\bar{\alpha}}{1-\bar{\alpha}}$. In this case, we get $\sum_{l=0}^{\infty} M_{2^l}^{\bar{\alpha}} < 1.1\frac{\bar{\alpha}}{1-\bar{\alpha}}$.

In the final case, $l_3 < 0$, we have $\sum_{l=0}^{\infty} M_{2^l}^{\bar{\alpha}} < \frac{0.66\bar{\alpha}^2}{(1-\bar{\alpha})^3} + \frac{0.15\bar{\alpha}}{(1-\bar{\alpha})^3}$. Since $\bar{\alpha} < 0.33$ in this case, we may use $\frac{\bar{\alpha}}{(1-\bar{\alpha})^2} < 0.75$ and $\frac{1}{(1-\bar{\alpha})^2} < 2.25$ to get the stated bound. $\qquad\square$

5.6 Improving the lookup cost

In this section we briefly describe an alternative to standard linear probing that improves the cost of lookups exponentially, without significantly changing the characteristics of linear probing. Update operations exhibit the same memory access pattern as before. Lookups perform jumps in memory, but still access only memory locations within a small interval – at most twice the length of the interval that would be inspected by the standard lookup procedure.

The idea is to order the keys of each maximal interval of occupied positions according to values of the hash function, in order to be able to do a *doubling search* during lookups. In other words, if there is a choice of more than one key to be placed at slot i then we choose the key having the hash value farthest from i (in the metric $(i - h(x)) \bmod r$). If there is more than one key with the most distant hash value, the smallest such key is stored at slot i. This invariant can be maintained during insertions and deletions at no asymptotic cost in running time, and the analysis of all operations stays the same.

Now consider a search for a key x, and assume for simplicity of exposition that r is a power of 2, and that the table is not full. Instead of searching for x sequentially we do a doubling search in the interval $h(x) + [r - 1]$ (which must contain x). For this to work we must argue that inspecting a location $(h(x) + i) \bmod r$ allows us to determine whether we should continue the search for x before or after. If $(h(x) + i) \bmod r$ is an empty location, it is clear that we must search before. By the invariant the same is true if location $(h(x) + i) \bmod r$ contains a key x' such that $h(x') \in h(x) + 1 + [i]$ or $h(x') = h(x) \wedge x' > x$. Otherwise, x cannot be in $h(x) + [i]$. The doubling search finds an interval $h(x) + [i, 2i]$ that contains x in case $x \in S$. Binary search is then applied on this interval. This means that any search that would take time l using standard linear probing now takes time $O(\log l)$. Specifically, the expected search time goes down to $O(\log \frac{1}{1 - \alpha})$.

5.7 Open problems

An immediate question is whether the dependence on α for linear probing with constant independence matches the dependence on α in the case of full independence, up to a constant factor. It is unclear whether 4 or even 3-wise independence can guarantee good expected performance for linear probing. If one could prove a sufficiently strong tail bound in a style of the bound from Lemma 5.2.1 it could be plugged into the framework of Section 5.4; the bound would have to be polynomially stronger than the bound that results from Chebyshev's inequality.

In general, the problem of finding practical, and provably good hash functions for a range of other important hashing methods remains unsolved. For example, cuckoo hashing [PR04] and its variants presently have no such functions. Also, if we consider the problem of hashing a set into $n / \log n$ buckets such that the number of keys in each bucket is $O(\log n)$ w.h.p., there is no known explicit class achieving this with function descriptions of $O(\log |U|)$ bits. Possibly, such families could be designed using efficient circuits, rather than a standard RAM instruction set.

Acknowledgments

Chapter 6

Deterministic load balancing and dictionaries in the parallel disk model

Abstract

We consider deterministic dictionaries in the parallel disk model, motivated by applications such as file systems. Our main results show that if the number of disks is moderately large (at least logarithmic in the size of the universe from which keys come), performance similar to the expected performance of randomized dictionaries can be achieved. Thus, we may avoid randomization by extending parallelism. We give several algorithms with different performance trade-offs. One of our main tools is a deterministic load balancing scheme based on expander graphs, that may be of independent interest.

6.1   Introduction

Storage systems for large data sets are increasingly *parallel*. There exist disk arrays consisting of more than a thousand disks[1], and *Network-Attached Storage* (NAS) solutions could in principle scale up to an arbitrary number of storage servers. A simple, feasible model for these kinds of situations is the parallel disk model [VS94]. In this model there are D storage devices, each consisting of an array of memory blocks with capacity for B *data items*; a data item is assumed to be sufficiently large to hold a pointer value or a *key* value. The performance of an algorithm is measured in the number of parallel I/Os, where one parallel I/O consists of retrieving (or writing) a block of B data items from (or to) each of the D storage devices.

The problem studied in this chapter is the design of *dictionaries*, that is, data structures storing a set of n keys from some bounded universe U, as well as "satellite" information associated with each key, supporting lookups of keys and dynamic updates to the key set. This is a fundamental and well-studied problem

---

[1]For example, the Hitachi TagmaStore USP1100 disk array can include up to 1152 disks, storing up to 32 petabytes.

in computer science. In the context of external memory, note that a dictionary can be used to implement the basic functionality of a file system: Let keys consist of a file name and a block number, and associate them with the contents of the given block number of the given file. Note that this implementation gives "random access" to any position in a file.

Most external memory algorithms for one disk can improve their performance by a factor of D in the parallel disk model using *striping*. For external memory data structures we can expect no such improvement, since at least one I/O is needed to answer on-line queries. For example, the query time of a B-tree in the parallel disk model is $\Theta(\log_{BD} n)$, which means that no asymptotic speedup is achieved compared to the one disk case unless the number of disks is very large, $D = B^{\omega(1)}$. Randomized dictionaries based on hashing support lookups and updates in close to 1 I/O per operation, as in the single disk case.

6.1.1 Our results and comparison with hashing

In this chapter we show a new kind of benefit by moving from one to many disks: Efficient randomized dictionaries may be replaced by deterministic ones of essentially the same efficiency. Besides the practical problem of giving an algorithm access to random bits, randomized solutions never give firm guarantees on performance. In particular, all hashing based dictionaries we are aware of may use $n/B^{O(1)}$ I/Os for a single operation in the worst case. In contrast, we give very good guarantees on the worst case performance of any operation. No previously known dynamic dictionary in a feasible model of computation has constant worst-case cost for all operations and linear space usage.

Randomized dictionaries

The most efficient randomized dictionaries, both in theory and practice, are based on hashing techniques. Much work has been devoted to the trade-off between time and space for such dictionaries, but in the context of this chapter we will only require that a dictionary uses linear space. While hashing algorithms were historically analyzed under the so-called *uniform hashing* assumption, most modern results are shown using explicit, and efficiently implementable, hash functions. In the context of external memory, the key requirement on a hash function is that its description should fit into internal memory. A reasonable assumption, made in the following, is that internal memory has capacity to hold $O(\log n)$ keys. This allows $O(\log n)$-wise independent hash functions, for which a large range of hashing algorithms can be shown to work well, see e.g. [SS90, SSS95].

There are dictionaries having performance that is, in an asymptotic sense, almost as good as one can hope for in a randomized structure. If one wants a randomized dictionary such that bounds on running times of its operations can be expressed in form $O(1)$, without interest in the actual constant, then a choice is the dictionary of [DGMP92], having lookup and update costs of $O(1)$ I/Os with high probability (the probability is $1 - O(n^{-c})$, where c can be chosen as any constant).

Method	Lookup I/Os	Update I/Os	Bandwidth	Conditions
[DGMP92]	$O(1)$	$O(1)$ whp.	-	-
Section 6.4.1	$O(1)$	$O(1)$	-	$D = \Omega(\log u)$
Hashing, no overflow	1 whp.	2 whp.	$O\left(\frac{BD}{\log n}\right)$	$BD = \Omega(\log n)$
Section 6.4.1	1	2	$O\left(\frac{BD}{\log n}\right)$	$D = \Omega(\log u)$ $B = \Omega(\log n)$
[PR04]	1	$O(1)$ am. exp.	$O(BD)$	-
[DGMP92] + trick	$1 + \epsilon$ avg. whp.	$2 + \epsilon$ avg. whp.	$O(BD)$	-
Section 6.4.3	$1 + \epsilon$ avg.	$2 + \epsilon$ avg.	$O(BD)$	$D = \Omega(\log u)$ $B = \Omega(\log n)$

Figure 6.1: Old and new results for linear space dictionaries with constant time per operation. The parameter ϵ can be chosen to be any positive constant; u denotes the size of the universe U. Update bounds take into account the cost of reading a block before it can be written, making 2 I/Os the best possible. Bounds that hold on average over all elements in the set are marked "avg." Where relevant (for dictionaries using close to 1 I/O), the bandwidth is also stated.

In our setting, having D parallel disks can be exploited by *striping*, that is, considering the disks as a single disk with block size BD. If BD is at least logarithmic in the number of keys, a linear space hash table (with a suitable constant) has no overflowing blocks with high probability. This is true even if we store associated information of size $O(BD/\log n)$ along with each key. Note that one can always use the dictionary to retrieve a pointer to satellite information of size BD, which can then be retrieved in an extra I/O. However, it is interesting how much satellite information can be returned in a single I/O. We call the maximum supported size of satellite data of a given method its *bandwidth*. Cuckoo hashing [PR04] can be used to achieve bandwidth $BD/2$, using a single parallel I/O, but its update complexity is only constant in the amortized expected sense.

In general, the *average* cost of an operation can be made arbitrarily close to 1, whp., by the following folklore trick: Keep a hash table storing all keys that do not collide with another key (in that hash table), and mark all locations for which there is a collision. The remaining keys are stored using the algorithm of [DGMP92]. The fraction of searches and updates that need to go to the dictionary of [DGMP92] can be made arbitrarily small by choosing the hash table size with a suitably large constant on the linear term. Note that the bandwidth for this method can be made $\Theta(BD)$ by allowing extra I/Os for operations on the keys in the second dictionary.

Our results

In this chapter we present *deterministic* and worst-case efficient results closely matching what can be achieved using hashing. All of our dictionaries use linear space. The first of our dictionaries achieves $O(1)$ I/Os in the worst case for both lookups and queries, without imposing any requirements on the value of B. A variation of this dictionary performs lookups in 1 I/O and updates in 2 I/Os, but requires that $B = \Omega(\log n)$; the bandwidth is $O(\frac{BD}{\log n})$, like in the comparable hashing result. We also give another dictionary that achieves the bandwidth of $O(BD)$ at the cost of relaxed operation performance; lookups take $1+\epsilon$ I/Os on average, and updates take $2+\epsilon$ I/Os on average. The worst-case cost is $O(\log n)$, as opposed to hashing where the worst-case cost is usually linear. Figure 6.1.1 shows an overview of main characteristics of all the mentioned dictionaries — the new ones and the comparable hashing-based structures.

All of our algorithms share features that make them suitable for an environment with many concurrent lookups and updates:

- There is no notion of an index structure or central directory of keys. Lookups and updates go directly to the relevant blocks, without any knowledge of the current data other than the size of the data structure and the size of the universe U.

- If we fix the capacity of the data structure and there are no deletions (or if we do not require that space of deleted items is reused), no piece of data is ever moved, once inserted. This makes it easy to keep references to data, and also simplifies concurrency control mechanisms such as locking.

The results in Figure 6.1.1 are achieved by use of an unbalanced expander graph of degree $O(\log u)$. While the existence of such a graph is known, the currently best explicit (i.e., computationally efficient) construction has degree polynomially larger than the optimal value [GUV07]. The presented dictionary structures may become a practical choice if and when practically efficient constructions of unbalanced expander graphs appear.

6.1.2 Motivation

File systems are by excellence an associative memory. This associative retrieval is implemented in most commercial systems through variations of B-trees. In a UNIX example, to retrieve a random block from a file (inode), one follows pointers down a tree with branching factor B; leaves hold pointers to the blocks of the file.

Since one does not need the additional properties of B-trees (such as range searching), a hash table implementation can be better. In theory, this can save an $O(\log_B n)$ factor. In practice, this factor is a small constant: in most settings it takes 3 disk accesses before the contents of the block is available. However, the file system is of critical importance to overall performance, and making just one disk read instead of 3 can have a tremendous impact. Furthermore, using a hash table can eliminate the overhead of translating the file name into an inode (which we have not counted above), since the name can be easily hashed as well.

Note that the above justification applies only to random accesses, since for sequential scanning of large files, the overhead of B-trees is negligible (due to caching). One may question the need for such random access. For algorithms on massive data sets it is indeed not essential. However, there are also critical applications of a more data-structural flavor. Popular examples include webmail or http servers. These typically have to retrieve small quantities of information at a time, typically fitting within a block, but from a very large data set, in a highly random fashion (depending on the desires of an arbitrary set of users). Arrays of disks are of course the medium of choice for such systems, so parallelism is readily available. Our results show how parallelism can, at least in theory, be used to provide an attractive alternative to B-trees in such settings.

From a theoretical perspective, we observe a trade-off between randomness and parallelism that has not been explored before. But our main motivation comes from looking at the potential applications. Randomization at the file-system level is an idea that is often frowned upon. For one, having to deal with expected running times adds unnecessary complications to a critical component of the operating system, and a potential for malfunction. More importantly, the file system often needs to offer a real-time guarantee for the sake of applications, which essentially prohibits randomized solutions, as well as amortized bounds.

6.1.3 Related work

The idea of using expander graphs for dictionaries appeared earlier in [BMRV02, ÖP02]. The results of [BMRV02] can be used to make a *static* dictionary (i.e. not supporting updates) in the parallel disk model, performing lookups in 1 I/O. The results of [ÖP02], which give a randomized structure for a serial RAM, can be modified to get a deterministic dynamic dictionary in a parallel setting [BHT05]. That dictionary has good *amortized* bounds on the time for updates, but analyzed in the *parallel disk head model* [AV88] (one disk with D read/write heads), which is stronger than the parallel disk model, and fails to model existing hardware. Additionally, the worst-case cost of updates was shown in [BHT05] to be linear in n. Our dictionaries have good worst-case performance, the I/O bounds on operations hold in the parallel disk model, and the methods are even simpler, in implementation as well as analysis, than the method of [ÖP02].

Other efforts towards efficient deterministic dictionaries (on a serial RAM) can be seen as derandomizations of hashing algorithms. However, the currently best methods need update time $n^{\Omega(1)}$ to achieve constant lookup time [Ruž08a, HMP01, Ruž08b].

6.1.4 Overview of chapter

In Section 6.2 we present definitions and notation to be used throughout the chapter. One of our main tools, a deterministic load balancing scheme is presented in Section 6.3. In section 6.4.1 we explain how to use the load balancing scheme to get an efficient dictionary in the parallel disk model. Section 6.4.2 presents another way of using expanders to get an efficient dictionary in the parallel disk model, in the static case where there are no updates. In Section 6.4.3

this scheme is dynamized to get a scheme that uses close to the optimal number of I/Os for operations, on average over all elements.

6.2 Preliminaries

An essential tool, common to all of our dictionary constructions is a class of expander graphs. There have been a number of definitions of expander graphs, some of them equivalent, and different notations have been used in the literature. The graphs that we use are bipartite. In a bipartite graph $G = (U, V, E)$, we may refer to U as the "left" part, and refer to V as the "right" part; a vertex belonging to the left (right) part is called a left (right) vertex. In our dictionary constructions, the left part corresponds to the universe U of keys, and the right part corresponds to the disk blocks of the data structure. A bipartite graph is called *left-d-regular* if every vertex in the left part has exactly d neighbors in the right part.

Definition 6.2.1. *A bipartite, left-d-regular graph $G = (U, V, E)$ is a (d, ε, δ)-expander if any set $S \subset U$ has at least $\min\left((1 - \varepsilon)d|S|, \ (1 - \delta)|V|\right)$ neighbors.*

Since expander graphs are interesting only when $|V| < d|U|$, some vertices must share neighbors, and hence the parameter ε cannot be smaller than $1/d$.

We introduce notation for the cardinalities of important sets: $u = |U|$, $v = |V|$, and $n = |S|$. The set of neighbors of a set $S \subset U$ is denoted by

$$\Gamma_G(S) = \{y \in V \mid (\exists x \in S) \ (x, y) \in E\} \ .$$

The subscript G will be omitted when it is understood, and we write $\Gamma(x)$ as a shorthand for $\Gamma(\{x\})$.

When G is an (d, ε, δ)-expander, then for any set $S \subset U$ such that $|S| < \frac{(1-\delta)v}{(1-\varepsilon)d}$ it holds that $|\Gamma(S)| \geq (1 - \varepsilon)d|S|$. It will be convenient to introduce another *notational* definition of expander graphs; this definition is used starting from Section 6.4.2 and until the end of the chapter.

Definition 6.2.2. *A bipartite, left d-regular graph $G = (U, V, E)$ is an (N, ε)-expander if any set $S \subset U$ of at most N left vertices has at least $(1 - \varepsilon)d|S|$ neighbors.*

There exist (d, ε, δ)-expanders with left degree $d = O(\log(\frac{u}{v}))$, for any v and positive constants ε, δ. If we wish that every subset of U having less than N elements expands "very well", that is, if we need a (N, ε)-expander it is possible to have $v = \Theta(Nd)$ (clearly it is a requirement that $v = \Omega(Nd)$).

For applications one needs an *explicit* expander, that is, an expander for which we can efficiently compute the neighbor set of a given node (in the left part). In the context of external memory algorithms, our requirement on an explicit construction is that the neighbor set can be computed without doing any I/Os (i.e., using only internal memory). No explicit constructions with the mentioned (optimal) parameters are known.

To make our algorithms work in the parallel disk model, we require an additional property of the expander graph: It should be *striped*. In a striped,

d-regular, bipartite graph there is a partition of the right side into d sets such that any left vertex has exactly one neighbor in each set of the partition. The notion of an explicit construction for striped expander graphs has the additional requirement that $\Gamma(x)$ for any given x, should be returned in form (i, j), where i is the index of the partition set and j is the index within that set. The results from random constructions mentioned previously hold even for striped random graphs. Unfortunately, most explicit constructions, in addition to not achieving optimal parameters, are also not striped.

6.3   Deterministic load balancing

We will consider d-choice load balancing using (unbalanced) bipartite expander graphs. Suppose there is an unknown set of n left vertices where each vertex has k items, and each item must be assigned to one of the *neighboring* right vertices (called "buckets"). We consider a natural greedy strategy for balancing the number of items assigned to each bucket. The assumption is that the set is revealed element by element, and the decision on where to assign the k items must be made on-line. The strategy is this: Assign the k items of a vertex one by one, putting each item in a bucket that currently has the fewest items assigned, breaking ties arbitrarily. The scheme allows multiple items belonging to a vertex to be placed in one neighboring bucket.

A special case of this load balancing scheme, where $k = 1$ and the bipartite graph is a random graph of left degree 2, was presented and analyzed in [ABKU99, BCSV00]. Tight bounds on the maximum load were given for the "heavily loaded case", showing that the deviation from the average load is $O(\log \log n)$ with a high probability. We now give an analogous result for a fixed (d, ε, δ)-expander. The scheme places a number of items in each bucket that is close to the average load of kn/v.

Lemma 6.3.1. *If $d > \frac{k}{1-\varepsilon}$ then after running the load balancing scheme using a (d, ε, δ)-expander, the maximum number of items in any bucket is bounded by $\frac{kn}{(1-\delta)v} + \log_{(1-\varepsilon)\frac{d}{k}} v$.*

Proof. Let $B(i)$ denote the number of buckets having *more* than i items, and let μ stand for $\frac{kn}{(1-\delta)v}$. We have that $B(\mu) < \frac{kn}{\mu} = (1 - \delta)v$. We will show that $(1 - \varepsilon)\frac{d}{k} \cdot B(\mu + i) \leq B(\mu + i - 1)$, for $i \geq 1$. Note that there are at least $B(\mu + i)/k$ left vertices that have placed an item in a bucket of load more than $\mu + i$ (after placement). Denoting the set of such left vertices by S_i, by the expansion property we have

$$|\Gamma(S_i)| \geq \min((1 - \varepsilon)d \cdot B(\mu + i)/k, \ (1 - \delta)v) \ .$$

Every vertex from S_i has all its neighboring buckets filled with more than $\mu + i - 1$ items, since the vertex was forced to put an item into a bucket of load larger than $\mu + i - 1$. If $|\Gamma(S_i)|$ was not smaller than $(1 - \delta)v$ then we would have $B(\mu + i - 1) \geq (1 - \delta)v$, which is a contradiction. As a result, $B(\mu + i) < (1 - \delta)v \cdot ((1 - \varepsilon)\frac{d}{k})^{-i}$, and it follows that $B(\mu + \log_{(1-\varepsilon)\frac{d}{k}} v) = 0$. □

It is not hard to observe that we actually get a bound of $\min(\frac{kn}{q} + \log_{(1-\varepsilon)\frac{d}{k}} q)$, where the minimum is over q ranging from 1 to $(1-\delta)v$. The simpler statement in the lemma is sufficient for the dictionary application, as we use expanders with v not too big.

6.4 Dictionaries on parallel disks

We consider dictionaries over a universe U of size u. It is sufficient to describe structures that support only lookups and insertions into a set whose size is not allowed to go beyond N, where the value of N is specified on initialization of the structure. This is because the dictionary problem is a *decomposable search problem*, so we can apply standard, worst-case efficient global rebuilding techniques (see [OvL81]) to get fully dynamic dictionaries, without an upper bound on the size of the key set, and with support for deletions. The main observations, assuming that we allow the number of disks to increase by a constant factor, are:

- The global rebuilding technique needed keeps two data structures active at any time, which can be queried in parallel.

- We can mark deleted elements without influencing the search time of other elements.

- We can make any constant number of parallel instances of our dictionaries. This allows insertions of a constant number of elements in the same number of parallel I/Os as one insertion, and does not influence lookup time.

The amount of space used and the number of disks increase by a constant factor compared to the basic structure. By the observations above, there is no time overhead. Deletions have the same worst case time bound as insertions.

6.4.1 Basic dictionary functionality

Without satellite information

Use a striped expander graph G with $v = N/\log N$, and an array of v (more elementary) dictionaries. The array is split across $D = d$ disks according to the stripes of G. The vertices from V represent indexes to the elements of the array. The dictionary implements the load balancing scheme described above, with $k = 1$. This gives a load of size $\Theta(\log N)$ on each bucket.

If the block size B is $\Omega(\log N)$, the contents of each bucket can be stored in a trivial way in $O(1)$ blocks. Thus, we get a dictionary with constant lookup time. By setting $v = O(N/B)$ sufficiently large we can get a maximum load of less than B, and hence membership queries take 1 I/O. The space usage stays linear.

The constraint $\Omega(\log N)$ is reasonable in many cases. Yet, even without making any constraints on B, we can achieve a constant lookup and insertion time by using an atomic heap [FW94, Hag98b] in each bucket. This makes the implementation more complicated; also, one-probe lookups are not possible in this case.

With satellite information

If the size of the satellite data is only a constant factor larger than the size of a key, we can increase v by a constant factor to allow that the associated data can be stored together with the keys, and can be retrieved in the same read operation. Larger satellite data can be retrieved in one additional I/O by following a pointer. By changing the parameters of the load balancing scheme to $k = d/2$ and $v = kN/\log N$, it is possible to accommodate lookup of associated information of size $O(BD/\log N)$ in one I/O. Technicalities on one-probe lookups — what exactly to write and how to merge the data — are given in the description of a one-probe static dictionary in Section 6.4.2

6.4.2 Almost optimal static dictionary

The static dictionary presented in this section is interesting in its own right: it offers one-probe lookups with good bandwidth utilization, uses linear space when $B = \Omega(\log n)$, and the construction complexity is within a constant factor from the complexity of sorting nd keys, each paired with some associated data. It is not optimal because it uses a bit more space when B is small and the construction procedure takes more time than the time it takes to sort the input, which would be fully optimal. The methods of this dictionary serve as a basis of the dynamic structure of the next section.

From now on, the graph $G = (U, V, E)$ is assumed to be an (N, ε)-expander, unless otherwise stated. We will work only with sets such that $n \leq N$.

Recall that the unique existential quantifier is denoted by $\exists!$. Let

$$\Phi_G(S) = \{y \in V \mid (\exists! x \in S)\ (x, y) \in E\}\ .$$

We call the elements of $\Phi_G(S)$ *unique neighbor* nodes. The following fact is known; it says that high expansion implies that any small set has many unique neighbors.

Lemma 6.4.1. *For any $S \subset U$ such that $|S| \leq N$, we have $|\Phi_G(S)| \geq (1 - 2\varepsilon)d|S|$.*

Proof. Suppose that $S = \{x_1, \dots, x_n\}$, $n \leq N$. Define the sets $T_k = \{x_1, \dots, x_k\}$, for $1 \leq k \leq n$. The sets T_k form a chain of subsets of S. We have that for any k:

$$\Gamma(T_{k+1}) \setminus \Gamma(T_k) = \Phi(T_{k+1}) \setminus \Phi(T_k)\ .$$

In the worst case, all the elements of $\Gamma(x_{k+1}) \cap \Gamma(T_k)$ will be in $\Phi(T_k)$. This leads to the inequality

$$|\Phi(T_{k+1})| \geq |\Phi(T_k)| - d + 2(|\Gamma(T_{k+1})| - |\Gamma(T_k)|)\ .$$

By induction, for all $k \leq n$ it holds that

$$2|\Gamma(T_k)| - |\Phi(T_k)| \leq k \cdot d\ .$$

Therefore $|\Phi(S)| \geq 2(1 - \varepsilon)dn - nd$. $\qquad\square$

Lemma 6.4.2. *Let $S' = \{x \in S \ : \ |\Gamma(x) \cap \Phi(S)| \geq (1-\lambda)d\}$, for a given $\lambda > 0$. Then $|S'| \geq (1 - \frac{2\varepsilon}{\lambda})n$.*

Proof. Suppose that $|S'| = n - k$. Let k^* be the largest integer that satisfies $k^*(1 - \lambda)d + (n - k^*)d > |\Phi(S)|$. It is easy to see that k^* is never smaller than $k = n - |S'|$. Using Lemma 6.4.1 to lower bound $|\Phi(S)|$ gives $k^* < \frac{2\varepsilon}{\lambda}n$. $\qquad\square$

For the following static and dynamic dictionary results, the stated numbers of used disks represent the minimum requirement for functioning of the dictionaries. The record of one key, together with some auxiliary data, is supposed to be distributed across $\frac{2}{3}d$ disks. For 1 I/O search to be possible, every distributed part of the record must fit in one block of memory. If the size of the satellite data is too large, more disks are needed to transfer the data in one probe. The degree of the graph does not change in that case, and the number of disks should be a multiple of d. Recall that we assume availability of a suitable expander graph construction such that $d = O(\log u)$.

Theorem 6.4.3. *Let σ denote the size in bits of satellite data of one element, and let d be the degree of the given (n, ε)-expander graph. In the parallel disk model there is a static dictionary storing a set of n keys with satellite data, such that lookups take one parallel I/O and the structure can be constructed deterministically in time proportional to the the time it takes to sort nd records with $\Theta(n\sigma)$ bits of satellite information in total. The exact usage of resources depends on the block size relative to the size of a key:*

  a) If $\Omega(\log n)$ keys can fit in one memory block, then the structure uses $2d$ disks and a space of $O(n(\log u + \sigma))$ bits;

  b) If the block size is smaller, then d disks are used and the space consumption is $O(n \log u \log n + n\sigma)$ bits.

The space usage in case (a) is optimal, up to a constant factor, when u is at least polynomially larger than n. When the universe is tiny, a specialized method is better to use, for example simple direct addressing. The space usage in case (b) is optimal when $\sigma > \log u \log n$.

Proof. Fix a value of ε that will always satisfy $1/d < \varepsilon < 1/6$; for concreteness we set $\varepsilon = 1/12$ (this imposes the restriction that $d > 12$). The data structure makes use of a striped (n, ε)-expander graph with left degree d and $v = O(nd)$. The main data is stored in an array A of v fields, where the size of a field depends on the case, (a) or (b). We will first explain the structure for case (b) in detail, then we will describe modifications made to optimize the space usage in case (a), and finally give the algorithm for construction of the structures.

Structure in case (b). Every field of A has size $\log n + \frac{3\sigma}{2d}$ bits (possibly rounded up to the nearest power of two). Given any $x \in U$ the set $\Gamma(x)$ is viewed as the set of indexes to the fields of A that *may* contain data about x. We will later describe how to accomplish that, for every $x \in S$, a 2/3 fraction of the fields referenced by $\Gamma(x)$ store parts of the associated record for x. When a

lookup is performed for a key x, all the fields of A pointed to by $\Gamma(x)$ are read into the internal memory in one parallel I/O. However, not all of them store data belonging to x. Deciding the correct ones could be done by storing a copy of the key in each of the fields of A that were assigned to it. Yet, a more space efficient solution is to use identifiers of $\log n$ bits, unique for each element of S. Upon retrieval of the blocks from disks, it is checked whether there exists an identifier that appears in more than half of the fields. If not, then clearly $x \notin S$; otherwise, the fields containing the majority identifier are merged to form a record of associated data. Note that no two keys from U can have more than εd common neighbors in V. Therefore, we know that the collected data belongs to x; there is no need for an additional comparison, or similar.

Structure in case (a). When the block size is reasonably large, we can avoid storing $\log n$ bits wide identifiers within the fields of A. We use two sub-dictionaries in parallel — one for pure membership queries and another for retrieval of satellite data. Half of $2d$ available disks is devoted to each dictionary. Checking membership in S is done using the dictionary from Section 6.4.1. Every stored key is accompanied by a small integer of $\log d$ bits, which we call its *head pointer*. By the assumption of the theorem for this case, $\Omega(\log n)$ such key-pointer pairs can fit in one block, thereby enabling one probe queries according to Section 6.4.1.

The retrieval structure is similar to the dictionary for case (b). The array A now has fields of size $\frac{3\sigma}{2d} + 4$ bits. Instead of "big" identifiers we choose to store succinct pointer data in every field; the fraction of an array field dedicated to pointer data will vary among fields. For every x we may introduce an ordering of the neighbor set $\Gamma(x)$ according to the stripes of V. That order implies an order of the fields of A assigned to a particular element. Each assigned field stores a *relative* pointer to the next field in the list; if the jth neighbor follows the ith neighbor in the list of assigned nodes, then the value $j - i$ is stored within $A(\Gamma(x, i))$, where $\Gamma(x, i)$ denotes the ith neighbor of x. The differences are stored in unary format, and a 0-bit separates this pointer data from the record data. The tail field just starts with a 0-bit. The entire space occupied by the pointer data is less than $2d$ bits per element of S; all the remaining space within fields is used on storing record data. Upon parallel retrieval of blocks from both dictionaries, we first check whether x is present in S. If it is, we use the head pointer to reconstruct the list and merge the satellite data.

Construction in $O(n)$ I/Os. Assigning $\lfloor \frac{2}{3}d \rfloor$ neighbors to each key from S is done using the properties of unique neighbor nodes. By setting $\lambda = 1/3$, according to Lemma 6.4.2 and the choice of ε, at least half of the elements of S have at least $\frac{2}{3}d$ neighbors unique to them. For each $x \in S'$ (with S' defined as in Lemma 6.4.2) any $\frac{2}{3}d$ unique neighbors are chosen to store its satellite data; the fields of unused nodes from $\Gamma(S') \cap \Phi(S)$ are labeled with an empty-field marker. The entire process of determining $\Phi(S)$, S', and filling the fields can be done in less than $c \cdot n$ parallel I/Os, for a constant c. The procedure is recursively applied to the set $S \setminus S'$, independently of the assignments done at the first level, because there is no intersection between the assigned neighbor set for S' and $\Gamma(S \setminus S')$.

The whole assignment procedure takes less than $c(n + n/2 + n/4 + \ldots) = O(n)$ I/Os.

Improving the construction. We keep the concept of making assignments using unique neighbor nodes, but change the procedure that realizes it. We assume that the input has a form of an array of records split across the disks, but with individual records undivided (this should be a standard representation). We will describe a procedure that is first applied to the input array, and then recursively applied to the array obtained by removing the elements of S' from the input array. Unlike the first version of the algorithm, the job is not completely finished for the elements of S' at the end of the procedure. The ouput of the procedure is an array of pairs of type (i, α_i), where α_i is data that is to be written to $A(i)$. This array will contain information only about the fields assigned to the elements of S'. Each time the procedure finishes execution, the output is appended to a global array, call it B.

The procedure starts by making an array of all pairs of type (x, y), $x \in S$, $y \in \Gamma(x)$. This array is sorted according to the second component of pairs, and then traversed to remove all sequences of more than one element that have equal values of the second components. This effectively leaves us a list of unique neighbor nodes, each paired with the (only) neighbor from the left side. By sorting this list of pairs according to the fist components, we get the elements of $\Phi(S) \cap \Gamma(x)$ grouped together, for each $x \in S$, and we are able to remove data about members of S that do not have enough unique neighbors. We now have a list of the elements of S', with associated lists of unique neighbors.

We sort the input array of records according to the dictionary keys. The resulting array is traversed simultaneously with the array of elements of S' (recall that this array is also sorted), allowing us to produce an array of pairs of type (i, α_i). The description of the main procedure is now finished.

When the contents of the array B is final (at the end of the recursion), it is sorted according to the first components of elements; this is the most expensive operation in the construction algorithm. Filling the array A is a straightforward task at this point. $\qquad\square$

6.4.3 Full bandwidth with $1 + \epsilon$ average I/O

The aforementioned static dictionary is not hard to dynamize in a way that gives fast average-case lookups and updates. We concentrate on the case (a) from Theorem 6.4.3. A slightly weaker result is possible in the more general case as well. In this section, the reader should be careful to distinguish between symbols ϵ and ε; ϵ is a parameter of operation performance, while ε is a parameter for expander graphs and its value will depend on the value of ϵ.

Theorem 6.4.4. *Let ϵ be an arbitrary positive value, and choose d, the degree of expander graphs, to be (a feasible value) larger than $6(1 + 1/\epsilon)$. Under the conditions of Theorem 6.4.3.a, there is a deterministic dynamic dictionary that provides the following performance: an unsuccessful search takes one parallel I/O, returning the associated data when a search is successful takes $1 + \epsilon$ I/Os averaged over all elements of S, updates run in $2 + \epsilon$ I/Os on average.*

124

Proof. As mentioned at the beginning of this section it is enough to describe a structure that supports only lookups and insertions into a set whose size is not allowed to go beyond N, where the value of N is specified on initialization of the structure. As in the static case, we use two sub-dictionaries. Since the first dictionary is already dynamic, modifications are made only to the dictionary that retrieves associated data. We choose ε so that $\frac{6}{d} < 6\varepsilon < 1/(1 + \frac{1}{\epsilon})$; the restriction on d was imposed to make this possible. Instead of just one array, now there are $l = \log N / \log \frac{1}{6\varepsilon}$ arrays of decreasing sizes: $A_1, A_2, \ldots, A_l$. The size of the array A_i is $(6\varepsilon)^{i-1}v$. Each array uses a different expander graph for field indexing; all expander graphs have the same left set U, the same degree d, but the size of each expander's right side equals the size of its corresponding array.

The insertion procedure works in a first-fit manner: for a given $x \in U$ find the first array in the sequence $(A_1, A_2, \ldots, A_l)$ in which there are $\frac{2}{3}d$ fields unique to x (at that moment). Using Lemma 6.4.2, it is not hard to check that the procedure is correct, i.e. a suitable place is always found. To briefly argue this, observe that for the resulting set of an insertion sequence, denote it by S, the array A_1 will hold the data for a superset of S' (with S' defined as in Lemma 6.4.2 and with respect to the first expander graph), and so forth. In the worst case, an insertion takes l reads and one write. However, any sequence of n insertions, $n \leq N$, requires n parallel writes and less than

$$n + (6\varepsilon)n + (6\varepsilon)^2 n + \ldots + (6\varepsilon)^l n$$

parallel read operations. The choice of ε implies the average of less than $1 + \epsilon$ reads. $\qquad\square$

6.5 Open problems

It is plausible that full bandwidth can be achieved with lookup in 1 I/O, while still supporting efficient updates. One idea that we have considered is to apply the load balancing scheme with $k = \Omega(d)$, recursively, for some constant number of levels before before relying on a brute-force approach. However, this makes the time for updates non-constant. It would be interesting if this construction could be improved.

Obviously, improved expander constructions would be highly interesting in the context of the algorithms presented in this chapter. It seems possible that practical and truly simple constructions could exist, e.g., a subset of d functions from some efficient family of hash functions.

Chapter 7

Near-Optimal Sparse Recovery in the L_1 norm

Abstract

We consider the *approximate sparse recovery problem*, where the goal is to (approximately) recover a high-dimensional vector $x \in \mathbb{R}^n$ from its lower-dimensional *sketch* $Ax \in \mathbb{R}^m$. Specifically, we focus on the sparse recovery problem in the L_1 norm: for a parameter k, given the sketch Ax, compute an approximation $\hat{x}$ of x such that the L_1 approximation error $\|x - \hat{x}\|_1$ is close to $\min_{x'} \|x - x'\|_1$, where x' ranges over all vectors with at most k terms. The sparse recovery problem has been subject to extensive research over the last few years. Many solutions to this problem have been discovered, achieving different trade-offs between various attributes, such as the sketch length, encoding and recovery times.

In this chapter we present a sparse recovery scheme which achieves close to optimal performance on virtually all attributes (see Figure 7.1). In particular, this is the first scheme that guarantees $O(k \log(n/k))$ sketch length, and near-linear $O(n \log(n/k))$ recovery time *simultaneously*. It also features low encoding and update times, and is noise-resilient.

7.1 Introduction

Over the recent years, a new approach for obtaining a succinct approximate representation of n-dimensional vectors (or signals) has been discovered. For any signal x, the representation is equal to Ax, where A is a $m \times n$ matrix. The vector Ax is often referred to as the *measurement vector* or *sketch* of x. Although m is typically much smaller than n, the sketch Ax contains plenty of useful information about the signal x.

The linearity of the sketching method is very convenient for a wide variety of applications. In the area of *data stream computing* [Mut03, Ind07], the vectors x are often very large, and cannot be represented explicitly; for example, x_i could denote the total number of packets with destination i passing through a network router. It is thus preferable to maintain instead the sketch Ax, under incremental updates to x. Specifically, if a new packet arrives, the corresponding coordinate of x is incremented by 1. This can be easily done if the sketching procedure is

linear. In the area of *compressed sensing* [CRT06a, Don06, TLW$^+$06, DDT$^+$08], the data acquisition itself is done using (analog or digital) hardware, which is capable of computing a dot product of the measurement vector and the signal at a unit cost. Other applications include breaking privacy of databases via aggregate queries [DMT07].

In this chapter, we focus on using linear sketches Ax to compute *sparse approximations* of x. Formally, we say that a vector y is *k-sparse* if it contains at most k non-zero entries. The goal is to find a vector $\hat{x}$ such that the ℓ_p approximation error $\|x - \hat{x}\|_p$ is at most $c > 0$ times the smallest possible ℓ_q approximation error $\|x - x'\|_q$, where x' ranges over all k-sparse vectors (we denote this type of guarantee by "$\ell_p \leq c\,\ell_q$"). Note that for any value of q, the error $\|x - \hat{x}\|_q$ is minimized when $\hat{x}$ consists of the k largest (in magnitude) coefficients of x.

The problem has been subject to an extensive research over the last few years, in several different research communities, including applied mathematics, digital signal processing and theoretical computer science. The goal of that research was to obtain encoding and recovery schemes with low probability of error (ideally, deterministic[1] schemes), short sketch lengths, low encoding, update and recovery times, good approximation error bounds and resilient to measurement noise. The current state of the art is presented in Figure 7.1. In the same figure we also show the best known bound for each of the aforementioned attributes of the algorithms (see the caption for further explanation).

Our result. The main result of this chapter is a very efficient sparse recovery scheme, with parameters as depicted in the last row of Figure 7.1. Up to the leading constants, the scheme achieves the best known bounds for: the error probability (our scheme is deterministic), sketch length, encoding and update times. Its decoding time is in general incomparable to the best prior bounds; however, it provides an improvement for k large enough. Finally, our scheme is resilient to noise (see Theorem 7.3.8 for the exact guarantee). The only drawback of our scheme is the $\ell_1 \leq C\ell_1$ error guarantee, which is known [CDD06] to be weaker than the $\ell_2 \leq \frac{C}{k^{1/2}}\ell_1$ guarantee achievable by some of the earlier schemes (although given that our scheme can be instantiated with $C = 1 + \epsilon$ for any $\epsilon > 0$, our guarantees are technically incomparable to those of [DM08, NT09]).

The efficiency with respect to many attributes makes our scheme an attractive option for sparse recovery problems. In particular, this is the first scheme that guarantees the $O(k \log(n/k))$ sketch length, and the near-linear decoding time *simultaneously*. Both attributes are of key importance: the sketch length determines the compression ratio (so any extra $\log n$ factor can reduce that ratio tenfold), while running times of $\Omega(nk)$ can quickly become prohibitive for n large enough (say, $n = 1000,000$). Measurement vector must have size $\Omega(k \log(n/k))$ [BIP09],[2] which means that the we have attained optimal sketch length.

---

[1] We use the term "deterministic" for a scheme in which one matrix A works for all signals x, and "randomized" for a scheme that generates a "random" matrix A which, for each signal x, works with probability $1 - 1/n$. However, "deterministic" does not mean "explicit" – we allow the matrix A to be constructed using the probabilistic method.

[2] The lower bound was proven for deterministic measurements. For randomized measurements, so far it was proven for the case $k = 1$.

Paper	R/D	Sketch length	Encoding time	Sparsity/Update time	Decoding time	Approx. error	N
[CCFC04, CM06]	R	$k\log^d n$	$n\log^d n$	$\log^d n$	$k\log^d n$	$\ell_2 \le C\ell_2$	
	R	$k\log n$	$n\log n$	$\log n$	$n\log n$	$\ell_2 \le C\ell_2$	
[CM05]	R	$k\log^d n$	$n\log^d n$	$\log^d n$	$k\log^d n$	$\ell_1 \le C\ell_1$	
	R	$k\log n$	$n\log n$	$\log n$	$n\log n$	$\ell_1 \le C\ell_1$	
[CRT06b]	D	$k\log(n/k)$	$nk\log(n/k)$	$k\log(n/k)$	LP	$\ell_2 \le \frac{C}{k^{1/2}}\ell_1$	Y
	D	$k\log^d n$	$n\log n$	$k\log^d n$	LP	$\ell_2 \le \frac{C}{k^{1/2}}\ell_1$	Y
[GSTV06]	D	$k\log^d n$	$n\log^d n$	$\log^d n$	$k\log^d n$	$\ell_1 \le C\log n\ell_1$	Y
[GSTV07]	D	$k\log^d n$	$n\log^d n$	$\log^d n$	$k^2\log^d n$	$\ell_2 \le \frac{\epsilon}{k^{1/2}}\ell_1$	
[GLR08] (k "large")	D	$k(\log n)^{d\log^{(3)} n}$	kn^{1-a}	n^{1-a}	LP	$\ell_2 \le \frac{C}{k^{1/2}}\ell_1$	
[BGI$^+$08]	D	$k\log(n/k)$	$n\log(n/k)$	$\log(n/k)$	LP	$\ell_1 \le C\ell_1$	Y
[DM08]	D	$k\log(n/k)$	$nk\log(n/k)$	$k\log(n/k)$	$nk\log\frac{n}{k}\log R$	$\ell_2 \le \frac{C}{k^{1/2}}\ell_1$	Y
[NT09]	D	$k\log(n/k)$	$nk\log(n/k)$	$k\log(n/k)$	$nk\log\frac{n}{k}\log R$	$\ell_2 \le \frac{C}{k^{1/2}}\ell_1$	Y
	D	$k\log^d n$	$n\log n$	$k\log^d n$	$n\log n\log R$	$\ell_2 \le \frac{C}{k^{1/2}}\ell_1$	Y
Best bounds per each column	D	$k\log(n/k)$	$n\log(n/k)$	$\log(n/k)$	$\min[k\log^d n, n\log n]$	$\ell_2 \le \frac{\epsilon}{k^{1/2}}\ell_1$	Y
This chapter	D	$k\log(n/k)$	$n\log(n/k)$	$\log(n/k)$	$n\log(n/k)$	$\ell_1 \le (1+\epsilon)\ell_1$	Y

Figure 7.1: Summary of the sparse recovery results. Virtually all references can be found at [Gro06]. All bounds ignore the $O(\cdot)$ constants. We also ignore other aspects, such as explicitness or universality of the measurement matrices. We present only the algorithms that work for arbitrary vectors x, while many other results are known for the case where the vector x itself is required to be k-sparse, e.g., see [TG05, DWB05, Don06, XH07]. The columns describe: citation; sketch type, deterministic or randomized; sketch length; time to compute Ax given x; time to update Ax after incrementing one of the coordinates of x; time to recover an approximation of x given Ax (see below); approximation guarantee (see below); does the algorithm tolerate noisy measurement vectors of the form $Ax+\nu$. The parameters $C > 1$, $d \ge 2$ and $a > 0$ denote some absolute constants, possibly different in each row. The parameter ϵ denotes any positive constant. We assume that $k < n/2$. In the decoding time column, LP=LP(n, m, T) denotes the time needed to solve a linear program defined by an $m \times n$ matrix A which supports matrix-vector multiplication (i.e., the encoding) in time T. Heuristic arguments indicate that LP$(n, m, T) \approx \sqrt{n}T$ if the interior-point method is employed. Some of the running times of the algorithms depend on the "precision parameter" R, which is always bounded from the above by $\|x\|_2$ if the coordinates of x are integers. It is known [CDD06] that "$\ell_2 \le \frac{c}{k^{1/2}}\ell_1$" implies "$\ell_1 \le (1+O(c))\ell_1$", and that it is impossible to achieve "$\ell_2 \le C\ell_2$" deterministically unless the number of measurements is $\Omega(n)$.

Our techniques. We use an adjacency matrix of an *unbalanced expander* as the encoding matrix A. Since such matrices can be very sparse (with only $O(\log(n/k))$ ones per column), the resulting scheme has very efficient encoding and update times. To make the scheme fully efficient, we also design a very fast recovery procedure that we call *Expander Matching Pursuit (EMP)* (Figure 7.2). The procedure roughly resembles the "greedy iterative approach" (a.k.a. *Orthogonal Matching Pursuit* [Tro04, TG05, NV09, DM08, NT09]), where the idea is to iteratively identify and eliminate "large" coefficients. However, in our procedure, the "large" coefficients are identified only at the beginning (in Step 1). In the remainder of the procedure, the choice of coordinates to iterate on is based on the structure of the expander matrix A, rather than the estimated magnitudes of coefficients.

We remark that the use of adjacency matrices of sparse random or expander graphs as encoding matrices for sparse approximation problems is not new; see, e.g., [CCFC04, CM06, CM05, GSTV06, GSTV07, XH07, Ind08, GLR08, BGI+08] for related work. However, all previous schemes were sub-optimal is some respects. In particular, the schemes of [CCFC04, CM06, CM05] provided only randomized guarantees and slightly worse measurement bounds; the sublinear-time algorithms of [GSTV06, GSTV07, Ind08] incurred polylogarithmic overhead in the number of measurements; the result of [XH07] was shown only for vectors x that are themselves k-sparse (or are slight generalizations of thereof); the matrices employed in [GLR08] had only sublinear, not logarithmic sparsity; and the decoding algorithm of [BGI+08] required solving a linear program, resulting in $\Omega(n^{3/2})$ running time.

Practicality of the algorithm and further developments. We have implemented a version of the EMP algorithm. As expected, the algorithm runs very fast. However, the number of measurements required by the algorithm to achieve correct recovery is somewhat suboptimal. In particular, we performed recovery experiments on random signed k-sparse signals of length n. For $k = 50$ and $n = 20000$, one typically needs about 5000 measurements to recover the signal. In comparison, the linear-programming-based recovery algorithm [BGI+08] requires only about 450 measurements to perform the same task[3].

Based on the ideas of the algorithm from this chapter, as well as from [NT09, BGI+08], we have developed an improved algorithm for the sparse recovery problem [BIR08]. The running time of the new algorithm, called *Sparse Matching Pursuit*, or *SMP*, is slightly higher (by a logarithmic factor) than of EMP, and has the same asymptotic bound on the number of required measurements. However, empirically, the algorithm performs successful recovery from a smaller number of measurements. In particular, for the instances described earlier, SMP typically needs about 1800 measurements. See [BIR08] for further empirical evaluation.

7.2 Preliminaries about expander graphs

An essential tool for our constructions are *unbalanced expander graphs*. Consider a bipartite graph $G = (U, V, E)$. We refer to U as the "left" part, and refer to

---

[3]For both algorithms we used randomly generated graphs with left degree equal to 20.

V as the "right" part; a vertex belonging to the left (right) part is called a left (right) vertex. In our constructions the left part will correspond to the set $\{1, 2, \ldots, n\}$ of coordinate indexes of vector x, and the right part will correspond to the set of row indexes of the measurement matrix. A bipartite graph is called *left-d-regular* if every vertex in the left part has exactly d neighbors in the right part.

Definition 7.2.1. *A bipartite, left-d-regular graph $G = (U, V, E)$ is an (s, d, ε)-expander if any set $S \subset U$ of at most s left vertices has at least $(1 - \varepsilon)d|S|$ neighbors.*

Using the probabilistic method one can show that there exist (s, d, ε)-expanders with $d = O(\log(|U|/s)/\varepsilon)$ and $|V| = O(s \log(|U|/s)/\varepsilon^2)$. For many applications one usually needs an *explicit* expander, that is, an expander for which we can efficiently compute the neighbor set of a given left vertex. No explicit constructions with the aforementioned (optimal) parameters are known. However, it is known [GUV07] how to explicitly construct expanders with left degree $d = O((\log |U|)(\log s)/\varepsilon)^{1+1/\alpha}$ and right set size $O(d^2 s^{1+\alpha})$, for any fixed $\alpha > 0$. In the remainder of this chapter, we will assume expanders with the optimal parameters. The set of neighbors of a set $S \subset U$ is denoted by

$$\Gamma_G(S) = \{v \in V \mid (\exists u \in S)\ (u, v) \in E\}\ .$$

The subscript G will be omitted when it is understood, and we write $\Gamma(u)$ as a shorthand for $\Gamma(\{u\})$. Recall that the unique existential quantifier is denoted by $\exists!$ – it can be read as "there exists a unique". Let

$$\Phi_G(S) = \{v \in V \mid (\exists! u \in S)\ (u, v) \in E\}\ .$$

We call the elements of $\Phi_G(S)$ *unique neighbor* nodes.

We will make use of the well-known fact that high expansion implies that any small set has many unique neighbors.

Lemma 7.2.2. *For any $S \subset U$ such that $|S| \leq s$, we have $|\Phi_G(S)| \geq (1 - 2\varepsilon)d|S|$.*

Lemma 7.2.3. *Let $S' = \{u \in S\ :\ |\Gamma(u) \cap \Phi(S)| \geq (1 - \lambda)d\}$, for a given $\lambda \geq 2\varepsilon$. Then $|S'| > (1 - \frac{2\varepsilon}{\lambda})|S|$.*

For the proofs of these two lemmas see Chapter 6. In the special case when $\lambda = 2\varepsilon$ the set S' contains at least one element.

7.3   Sparse recovery

We consider linear sketches of form Ax, where A is an $m \times n$ adjacency matrix of a (s, d, ε)-expander G, where $s = 4k$, $d = O(\frac{1}{\varepsilon} \log \frac{n}{k})$, $m = O(\frac{k}{\varepsilon^2} \log \frac{n}{k})$ and $\varepsilon < 1/16$. We will consider the general case of noisy measurements. Let $c = Ax + \nu$ be a sketch vector contaminated with noise ν.

We use K to denote the set of k indexes of coordinates of x having k largest magnitudes (ties are broken arbitrarily). In other words, x_K is a best k-term approximation to x. For a coordinate subset I we will write x_I to denote the vector obtained from x by zeroing-out all entries outside of I.

Expander Matching Pursuit

1. Compute $x^* = x^*(c)$ such that for any $i = 1 \ldots n$:

$$x_i^*(c) = \text{median}(c_{j_1}, c_{j_2}, \ldots, c_{j_d}) \ , \qquad (7.1)$$

   where $\{j_1, \ldots, j_d\} = \Gamma(i)$;

2. Find the set I consisting of $2k$ indexes of the coordinates of x^* of highest magnitudes, breaking ties arbitrarily;

3. Find the set $\bar{I}$ being the smallest-size superset of I such that

$$(\forall i \in \{1, 2, \ldots, n\} \setminus \bar{I}) \ |\Gamma(i) \cap \Gamma(\bar{I})| \leq 2\varepsilon d \ ; \qquad (7.2)$$

   (we provide a more detailed description of this step later in this section)

4. $\hat{x} \leftarrow \mathbf{0}$;

5. $j \leftarrow 1$; $\quad I_j \leftarrow \bar{I}$;

6. Repeat the following sequence of steps until $I_j = \emptyset$:

   (a) Find $I_j' = \{i \in I_j \ : \ |\Gamma(i) \cap \Phi(I_j)| \geq (1 - 2\varepsilon)d\}$;
   (b) $x^* \leftarrow x^*(c)$;
   (c) $\hat{x}_{I_j'} \leftarrow x_{I_j'}^*$;
   (d) $c \leftarrow c - Ax_{I_j'}^*$;
   (e) $I_{j+1} \leftarrow I_j \setminus I_j'$; $\quad j \leftarrow j + 1$;

Figure 7.2: Recovery algorithm

7.3.1 Algorithm

The outline of the recovery algorithm is given in Figure 7.2. Note that the nonzero components of the approximation $\hat{x}$ are confined to the set $\bar{I}$. The set $\bar{I}$ can be computed as as follows. First, observe that the set $\bar{I}$ is uniquely defined. The following claim establishes that $|\bar{I}| < 4k$.

Claim 7.3.1. *Let γ be a constant value larger than ε. Suppose that $I \subset \{1, 2, \ldots, n\}$ is a given set of coordinate positions, with $|I| \leq (1 - \frac{\varepsilon}{\gamma})s$. Let J be the smallest-size superset of I with the property that*

$$(\forall j \in \{1, 2, \ldots, n\} \setminus J) \ |\Gamma(j) \cap \Gamma(I)| \leq \gamma d \ . \qquad (7.3)$$

The size of the set J is smaller than $(1 - \frac{\varepsilon}{\gamma})^{-1}|I|$.

Proof. Because J is the smallest-size superset of I satisfying (7.3), it follows that there exists a sequence of indexes $(j_1, \ldots, j_p)$, with $p = |J| - |I|$, such that

$\{j_1, \ldots, j_p\} = J \setminus I$ and

$$|\Gamma(j_{k+1}) \cap \Gamma(I \cup \{j_1, \ldots, j_k\})| > \gamma d \ .$$

Assume by contradiction that $|J| \geq (1 - \frac{\varepsilon}{\gamma})^{-1}|I|$. Let $l = (1 - \frac{\varepsilon}{\gamma})^{-1}|I| - |I|$, and define $J' = I \cup \{j_1, \ldots, j_l\}$. We have that $d|J'| - |\Gamma(J')| > \gamma dl$, and then $d(|J'| - \gamma l) > |\Gamma(J')| \geq (1 - \varepsilon)d|J'|$ (the last inequality is true because $|J'| \leq s$). We see that $\gamma l < \varepsilon|J'|$, and thus $(\gamma - \varepsilon)|J'| < \gamma|I|$, which contradicts the earlier assumption about the size of J'. □

We now discuss the procedure for finding $\bar{I}$, together with the running time analysis. Initially, we let $\bar{I}$ be equal to I. The algorithm maintains a priority queue over the set $\{1, 2, \ldots, n\} \setminus \bar{I}$, with the priority of element i being $|\Gamma(i) \cap \Gamma(\bar{I})|$. Each vertex from $V \setminus \Gamma(\bar{I})$ will have an associated list of references to the elements of the priority queue that have it as a neighbor. When a vertex from V enters the set $\Gamma(\bar{I})$ the priorities of the corresponding elements are updated. Elements whose priorities become higher than $2\varepsilon d$ are added to the set $\bar{I}$. The priority queue can be implemented as an array of $2\varepsilon d$ linked lists, so that every required operation runs in constant time. The entire process takes a time of $O(nd)$, since we can attribute a unit cost of work to every edge of G. The algorithm uses $O(n)$ words of storage (in addition to the space used to store the matrix A). A more space-efficient (but slightly slower) algorithm is described in the appendix.

Total running time of step 6 of the algorithm is $O(kd)$. The procedure that performs step 6.a uses a similar method with a priority queue, only that here elements get extracted from the set, and priorities are decreasing. This part uses $O(kd)$ words of storage.

In the remainder of this section we will focus on proving the approximation bounds. We start from technical lemmas providing guarantees for the initial approximation vector x^*. Then we give the proof of the approximation guarantee.

7.3.2 Technical lemmas

The statement of the following lemma may look somewhat unintuitive, due to the fact that its formulation needs to allow a proof by induction. More intuitive error bounds for estimates x_i^* will be presented in Theorem 7.3.3.

Lemma 7.3.2. *Let $I, J, L, M \subset \{1, 2, \ldots, n\}$ be given sets of coordinate positions such that*

- $|I| \leq s/2$;

- I, J, L, M *are mutually disjoint, and* $I \cup J \cup L \cup M = \{1, 2, \ldots, n\}$;

- $(\forall l \in L)(|\Gamma(l) \cap \Gamma(I)| \leq 2\varepsilon d)$;

- $(\forall i \in I)(|\Gamma(i) \cap \Gamma(M)| \leq \alpha d)$, *where* $\alpha < 1/2 - 2\varepsilon$.

There exist a chain of subsets of I and a family of disjoint subsets of J, which we respectively write as $I = I_0 \supset I_1 \supset \ldots \supset I_q$ and $\{J_0, J_1, \ldots, J_q\}$ [4], satisfying the following two properties:

$$\|x_I - x_I^*\|_1 \leq (1/2 - 2\varepsilon - \alpha)^{-1} \Big(2\varepsilon \|x_I\|_1 + \frac{1}{d}\|(A\,x_L)_{\Gamma(I)}\|_1 +$$
$$+ \frac{1}{d}\sum_{k=0}^{q} \|(A\,x_{J_k})_{\Gamma(I_k)}\|_1 + \frac{1}{d}\|\nu_{\Gamma(I)}\|_1 \Big)\ ,$$

and $(\forall j \in J_k)(|\Gamma(j) \cap \Gamma(I_k)| \leq 2\varepsilon d)$, for $0 \leq k \leq q$.

Proof. We will prove the claim by induction on the size of the set I. Suppose first that $|I| = 1$, that is $I = \{i\}$ for some i. In this case we have that $q = 0$ and $J_0 = J$. Consider the multiset $\phi = \{c_{v_1}, c_{v_2}, \ldots, c_{v_l}\}$ such that v_k are the indexes from $\Gamma(i) \setminus M$ (therefore $l \geq (1-\alpha)d$). The estimate x_i^* will have a rank between $d/2 - (d - l)$ and $d/2$ with respect to the multiset ϕ (the value x_i^* need not be equal to one of the elements of ϕ). In other words, at least $(1/2 - \alpha)d$ elements of ϕ are not larger than x_i^*, and at at least $(1/2 - \alpha)d$ elements of ϕ are not smaller than x_i^*. Therefore in any case ($x_i < x_i^*$ or $x_i > x_i^*$) we have that

$$(1/2 - \alpha)d\,|x_i - x_i^*| \leq \|(A\,x_{\{i\}} - c)_{\Gamma(i)\setminus M}\|_1 \leq \|(A\,x_{L \cup J_0} + \nu)_{\Gamma(I)}\|_1$$
$$\leq \|(A\,x_L)_{\Gamma(I)}\|_1 + \|(A\,x_{J_0})_{\Gamma(I)}\|_1 + \|\nu_{\Gamma(I)}\|_1\ ,$$

which proves the claim for the case $|I| = 1$.

Now suppose that $|I| > 1$ and that the claim is true for index sets of smaller sizes. Let $J^* = \{j \in J\ :\ |\Gamma(j) \cap \Gamma(I)| > 2\varepsilon d\}$. Because of the expansion property of the graph G and the condition that $|I| \leq s/2$, the size of J^* can be at most $|I|$. Let

$$I' = \{i \in I\ :\ |\Gamma(i) \cap \Phi(I \cup J^*)| \geq (1 - 2\varepsilon)d\}\ .$$

We will have that for any $i \in I'$ the influence of other entries from $x_{I \cup J^*}$ to the estimate x_i^* is relatively minor. Since all elements from J^* have less than $(1 - 2\varepsilon)d$ unique neighbor nodes with respect to the set $I \cup J^*$, from Lemma 7.2.3 we conclude that $|I'| \geq 1$. Let $I_1 = I \setminus I'$, and apply the induction hypothesis to the set I_1, with I' and $J \setminus J^*$ merged with the set L. Let the returned quasi-partition of J^* be $\{J_0', J_1', \ldots, J_{q-1}'\}$. We make assignments $J_{k+1} = J_k'$, $0 \leq k < q$, and $J_0 = J \setminus J^*$.

By the triangle inequality and the induction hypothesis we have that

$$\|x_I - x_I^*\|_1 \leq \|x_{I'} - x_{I'}^*\|_1 + (1/2 - 2\varepsilon - \alpha)^{-1} \Big(2\varepsilon\|x_{I_1}\|_1 + \quad (7.4)$$
$$+ \frac{1}{d}\|(A\,x_{I' \cup L \cup J_0})_{\Gamma(I_1)}\|_1 +$$
$$+ \frac{1}{d}\sum_{k=1}^{q} \|(A\,x_{J_k})_{\Gamma(I_k)}\|_1 + \frac{1}{d}\|\nu_{\Gamma(I_1)}\|_1 \Big)\ .$$

---

[4] Some of the subsets may be empty; thus we cannot call this family a partition of J.

To upper-bound $\|(A\,x_{I'})_{\Gamma(I_1)}\|_1$ we use the fact that with respect to the graph G each $i \in I'$ has at most $2\varepsilon d$ adjacent nodes shared with the elements of I_1.

$$\|(A\,x_{I'})_{\Gamma(I_1)}\|_1 \leq \sum_{i \in I'} \|(A\,x_{\{i\}})_{\Gamma(I_1)}\|_1 \leq \sum_{i \in I'} 2\varepsilon d|x_i| = 2\varepsilon d\|x_{I'}\|_1 \ .$$

Now we will analyse estimation error for coordinates in I'. For any $i \in I'$ consider the multiset $\phi^{(i)} = \{c_{v_1}, c_{v_2}, \ldots, c_{v_l}\}$ such that v_k are the indexes from $\Gamma(i) \cap \Phi(I \cup J^* \cup M)$ (therefore $l \geq (1 - 2\varepsilon - \alpha)d$). The estimate x_i^* will have a rank between $d/2 - (d - l)$ and $d/2$ with respect to the multiset $\phi^{(i)}$ (the value x_i^* need not be equal to one of the elements of $\phi^{(i)}$). In other words, at least $(1/2 - 2\varepsilon - \alpha)d$ elements of $\phi^{(i)}$ are not larger than x_i^*, and at at least $(1/2 - 2\varepsilon - \alpha)d$ elements of $\phi^{(i)}$ are not smaller than x_i^*. Therefore,

$$\|(A\,x_{\{i\}} - c)_{\Gamma(i) \cap \Phi(I \cup J^* \cup M)}\|_1 \geq (1/2 - 2\varepsilon - \alpha)d\,|x_i - x_i^*| \ .$$

By aggregating over all $i \in I'$ we get

$$
\begin{aligned}
(1/2 - 2\varepsilon - \alpha)d\,\|x_{I'} - x_{I'}^*\|_1 \ &\leq \ \|(A\,x_{I'} - c)_{\Gamma(I') \cap \Phi(I \cup J^* \cup M)}\|_1 \leq \\
&\leq \ \|(A\,x_{L \cup J_0} + \nu)_{\Gamma(I') \cap \Phi(I \cup J^* \cup M)}\|_1 \leq \\
&\leq \ \|(A\,x_L)_{\Gamma(I') \cap \Phi(I \cup J^* \cup M)}\|_1 + \\
&\quad + \|(A\,x_{J_0})_{\Gamma(I') \cap \Phi(I \cup J^* \cup M)}\|_1 + \\
&\quad + \|\nu_{\Gamma(I') \cap \Phi(I \cup J^* \cup M)}\|_1 \ .
\end{aligned}
$$

These terms are substituted into (7.4). Observe that the following four inequalities hold

$$
\begin{aligned}
\|(A\,x_L)_{\Gamma(I') \cap \Phi(I \cup J^* \cup M)}\|_1 + \|(A\,x_L)_{\Gamma(I_1)}\|_1 \ &\leq \ \|(A\,x_L)_{\Gamma(I)}\|_1 \ , \\
\|(A\,x_{J_0})_{\Gamma(I') \cap \Phi(I \cup J^* \cup M)}\|_1 + \|(A\,x_{J_0})_{\Gamma(I_1)}\|_1 \ &\leq \ \|(A\,x_{J_0})_{\Gamma(I)}\|_1 \ , \\
\|\nu_{\Gamma(I') \cap \Phi(I \cup J^* \cup M)}\|_1 + \|\nu_{\Gamma(I_1)}\|_1 \ &\leq \ \|\nu_{\Gamma(I)}\|_1 \ , \\
2\varepsilon\|x_{I_1}\|_1 + 2\varepsilon\|x_{I'}\|_1 \ &\leq \ 2\varepsilon\|x_I\|_1 \ .
\end{aligned}
$$

This proves the claimed bound on $\|x_I - x_I^*\|_1$. $\qquad\square$

The following theorem is a simple consequence of Lemma 7.3.2.

Theorem 7.3.3. *Let $I \subset \{1, 2, \ldots, n\}$ be a given set of coordinate positions, with $|I| \leq s/2$. Suppose that $\{J, M\}$ is a partition of the set of remaining coordinates $\{1, 2, \ldots, n\} \setminus I$ where the set M satisfies*

$$(\forall i \in I)(|\Gamma(i) \cap \Gamma(M)| \leq \alpha d) \ ,$$

with $\alpha < 1/2 - 2\varepsilon$. It holds that

$$\|x_I - x_I^*\|_1 \leq (1/2 - 2\varepsilon - \alpha)^{-1}\left(2\varepsilon\|x_{I \cup J}\|_1 + \tfrac{1}{d}\|\nu_{\Gamma(I)}\|_1\right) \ .$$

We will now state a lemma for a special case when it is known that all elements of I have "many" unique neighbor nodes, and the remaining elements have a well determined structure of intersections with the neighbors of I.

Lemma 7.3.4. *Let $\{I, J, L, M\}$ be a given partition of $\{1, 2, \ldots, n\}$ with the following properties:*

- $|I| \leq s/2$ and $(\forall i \in I)(|\Gamma(i) \cap \Phi(I \cup M)| \geq (1 - \alpha)d)$, where $\alpha < 1/2$;

- $(\forall l \in L)(|\Gamma(l) \cap \Gamma(I)| \leq \beta d)$;

- *Let* $\Delta = \frac{1}{d}\|(A\,x_J + \nu)_{\Gamma(I) \cap \Phi(I \cup M)}\|_1$.

It holds that $\|x_I - x_I^*\|_1 \leq (1/2 - \alpha)^{-1}(\beta\|x_L\|_1 + \Delta)$.

Proof. The proof is a simpler version of the previous proofs. For any $i \in I$ consider the multiset $\phi^{(i)} = \{c_{v_1}, c_{v_2}, \ldots, c_{v_l}\}$ such that v_k are the indexes from $\Gamma(i) \cap \Phi(I \cup M)$ (therefore $l \geq (1-\alpha)d$). The estimate x_i^* will have a rank between $d/2 - (d - l)$ and $d/2$ with respect to the multiset $\phi^{(i)}$. In other words, at least $(1/2 - \alpha)d$ elements of $\phi^{(i)}$ are not larger than x_i^*, and at at least $(1/2 - \alpha)d$ elements of $\phi^{(i)}$ are not smaller than x_i^*. Therefore in any case ($x_i < x_i^*$ or $x_i > x_i^*$) we have that

$$\|(A\,x_{\{i\}} - c)_{\Gamma(i) \cap \Phi(I \cup M)}\|_1 \geq (1/2 - \alpha)d\,|x_i - x_i^*| \ .$$

By aggregating over all $i \in I'$ we get that

$$
\begin{aligned}
(1/2 - \alpha)d\,\|x_I - x_I^*\|_1 \quad &\leq \quad \|(A\,x_I - c)_{\Gamma(I) \cap \Phi(I \cup M)}\|_1 \leq \\
&\leq \quad \|(A\,x_{L \cup J} + \nu)_{\Gamma(I) \cap \Phi(I \cup M)}\|_1 \leq \\
&\leq \quad \|(A\,x_L)_{\Gamma(I)}\|_1 + \|(A\,x_J + \nu)_{\Gamma(I) \cap \Phi(I \cup M)}\|_1 \ .
\end{aligned}
$$

Similarly as earlier we observe that

$$
\begin{aligned}
\|(A\,x_L)_{\Gamma(I)}\|_1 \quad &\leq \quad \sum_{l \in L}\|(A\,x_{\{l\}})_{\Gamma(I)}\|_1 = \sum_{l \in L}|\Gamma(l) \cap \Gamma(I)| \cdot |x_l| \leq \\
&\leq \quad \sum_{l \in L}\beta d|x_l| = \beta d\|x_L\|_1 \ .
\end{aligned}
$$

The claimed bound on $\|x_I - x_I^*\|_1$ now easily follows. $\qquad\square$

The following corollary is an obvious consequence of Theorem 7.3.3.

Corollary 7.3.5. *Suppose that noise ν is zero. For any $\lambda \geq 1$,*

$$\left|\left\{i \in \{1, 2, \ldots, n\} \ : \ |x_i - x_i^*| > \frac{8\varepsilon}{1 - 4\varepsilon}\frac{\lambda}{s}\|x\|_1\right\}\right| < \frac{s}{2\lambda} \ .$$

7.3.3 Approximation guarantees

In this section we finish the analysis of the approximation error. We start by showing that the set $\bar{I}$ contains all "important" coefficients of the vector x.

Lemma 7.3.6. $\|x - x_{\bar{I}}\| \leq \frac{1 - 8\varepsilon}{1 - 16\varepsilon}\|x - x_K\|_1 + \frac{4}{(1 - 16\varepsilon)d}\|\nu\|_1$.

Proof. Let $K_1 = K \setminus \bar{I}$. If $K_1 = \emptyset$ then the claim is clearly true. In general we need to show that $\|x_{K_1}\|_1$ is not much larger than $\|x_{\bar{I}\setminus K}\|_1$. Let $I' = \{i \in \bar{I} : |\Gamma(i) \cap \Phi(\bar{I})| \geq (1-4\varepsilon)d\} \setminus K$. According to Lemma 7.2.3, it is $|I'| \geq \frac{1}{2}|\bar{I}| - |K \cap \bar{I}| \geq k - |K \cap \bar{I}| = |K_1|$. Since every coordinate of $x_{K_1}^*$ is not smaller than any coordinate of $x_{I'}^*$ we see that $\|x_{I'}^*\|_1 \geq \|x_{K_1}^*\|_1$. Hence, $\|x_{I'}\|_1 + \|x_{I'} - x_{I'}^*\|_1 \geq \|x_{K_1}\|_1 - \|x_{K_1} - x_{K_1}^*\|_1$, and so $\|x_{K_1}\|_1 - \|x_{I'}\|_1 \leq \|x_{I'} - x_{I'}^*\|_1 + \|x_{K_1} - x_{K_1}^*\|_1$.

An upper bound on $\|x_{I'} - x_{I'}^*\|_1$ follows from Lemma 7.3.4, (in the context of Lemma 7.3.4 we have $\alpha = 4\varepsilon$, $\beta = 2\varepsilon$, and $\Delta = \frac{1}{d}\|\nu_{\Gamma(I')\cap\Phi(I)}\|_1$). Therefore, $\|x_{I'} - x_{I'}^*\|_1 \leq (1/2 - 4\varepsilon)^{-1}(2\varepsilon\|x - x_{\bar{I}}\|_1 + \frac{1}{d}\|\nu\|_1)$. To bound $\|x_{K_1} - x_{K_1}^*\|_1$ we apply Theorem 7.3.3, which gives $\|x_{K_1} - x_{K_1}^*\|_1 \leq (1/2 - 4\varepsilon)^{-1}(2\varepsilon\|x - x_{\bar{I}}\|_1 + \frac{1}{d}\|\nu\|_1)$. Combining the obtained inequalities we get that

$$
\begin{aligned}
\|x - x_{\bar{I}}\| &\leq \|x - x_K\|_1 + \|x_{K_1}\|_1 - \|x_{I'}\|_1 \\
&\leq \|x - x_K\|_1 + \|x_{I'} - x_{I'}^*\|_1 + \|x_{K_1} - x_{K_1}^*\|_1 \\
&\leq \|x - x_K\|_1 + \frac{8\varepsilon}{1-8\varepsilon}\|x - x_{\bar{I}}\|_1 + \frac{4}{(1-8\varepsilon)d}\|\nu\|_1 \ ,
\end{aligned}
$$

which implies the claimed bound. $\square$

Lemma 7.3.7. *Suppose that $(\Delta_{ij})_{1\leq i<j\leq p}$ is a sequence of real values that satisfy for each i*

$$
\sum_{j=i+1}^{p} \Delta_{ij} \leq \rho(\Delta_i + \Delta_{1i} + \ldots + \Delta_{i-1\,i}) \ ,
$$

where ρ and Δ_i are some constants, with $0 < \rho < 1$. Let Δ denote $\sum_i \Delta_i$. The following inequality holds:

$$
\sum_{i\geq 1}\sum_{j>i} \Delta_{ij} \leq \frac{\rho}{1-\rho}\Delta \ .
$$

Proof. We have that

$$
\sum_{i\geq 1}\sum_{j>i} \Delta_{ij} \leq \sum_{i\geq 1}\rho\Big(\Delta_i + \sum_{k=1}^{i-1}\Delta_{ki}\Big) = \rho\Delta + \rho\sum_{l\geq 1}\sum_{m>l}\Delta_{lm} \ .
$$

As a result, $(1-\rho)\sum_{i\geq 1}\sum_{j>i}\Delta_{ij} \leq \rho\Delta$. $\square$

Theorem 7.3.8. *Given a vector $c = Ax + \nu$, the algorithm returns approximation vector $\hat{x}$ satisfying*

$$
\|x - \hat{x}\|_1 \leq \frac{1-4\varepsilon}{1-16\varepsilon}\|x - x_K\|_1 + \frac{6}{(1-16\varepsilon)d}\|\nu\|_1 \ .
$$

where K is the set of the k largest (in magnitude) coordinates of x.

Proof. Let $R = \{1, 2, \ldots, n\} \setminus \bar{I}$ and $\Delta_j = \frac{1}{d}\|(A\,x_R + \nu)_{\Gamma(I'_j)\cap\Phi(I_j)}\|_1$, $j \geq 1$. Denoting $\Delta = \sum_j \Delta_j$, we have that

$$
\Delta \leq \frac{1}{d}\|(A\,x_R + \nu)_{\bar{I}}\|_1 \leq 2\varepsilon\|x_R\|_1 + \frac{1}{d}\|\nu\|_1 \ .
$$

137

When we write $x^*_{I'_j}$ it formally means $x^*_{I'_j}(c^j)$, where c^j is the value of vector c at the beginning of the jth iteration of step 6 of the algorithm. By Lemma 7.3.4 it is $\|x_{I'_1} - x^*_{I'_1}\|_1 \leq (1/2 - 2\varepsilon)^{-1}\Delta_1$. Let $\nu' = A(x_{I'_1} - x^*_{I'_1})$ and $\Delta_{1j} = \frac{1}{d}\|\nu'_{\Gamma(I'_j) \cap \Phi(I_j)}\|_1$, for $j \geq 2$. We have that

$$\sum_{j \geq 2} \Delta_{1j} \leq \frac{1}{d}\|(A(x_{I'_1} - x^*_{I'_1}))_{\Gamma(I_2)}\|_1 \leq 2\varepsilon\|x_{I'_1} - x^*_{I'_1}\|_1 \leq 2\varepsilon(1/2 - 2\varepsilon)^{-1}\Delta_1 \ .$$

To bound $\|x_{I'_2} - x^*_{I'_2}\|_1$ in the second step of the algorithm we will again use Lemma 7.3.4. Let $x' = x - x_{I'_1}$. For the second step we can write that $c = Ax' + \nu + \nu'$, so $\nu + \nu'$ is viewed as noise. Since $x_{I'_2} = (x')_{I'_2}$, through Lemma 7.3.4 we get that $\|x_{I'_2} - x^*_{I'_2}\|_1 \leq (1/2 - 2\varepsilon)^{-1}(\Delta_2 + \Delta_{12})$. In general, let $\Delta_{ij} = \frac{1}{d}\|(A(x_{I'_i} - x^*_{I'_i}))_{\Gamma(I'_j) \cap \Phi(I_j)}\|_1$, for $j > i \geq 1$. Similarly as before we find that

$$\|x_{I'_j} - x^*_{I'_j}\|_1 \leq (1/2 - 2\varepsilon)^{-1}(\Delta_j + \Delta_{1j} + \ldots + \Delta_{j-1\,j}) \ .$$

Further,

$$\begin{aligned}
\sum_{l > j} \Delta_{jl} &\leq \frac{1}{d}\|(A(x_{I'_j} - x^*_{I'_j}))_{\Gamma(I_{j+1})}\|_1 \leq 2\varepsilon\|x_{I'_j} - x^*_{I'_j}\|_1 \leq \\
&\leq 2\varepsilon(1/2 - 2\varepsilon)^{-1}(\Delta_j + \Delta_{1j} + \ldots + \Delta_{j-1\,j}) \ .
\end{aligned}$$

Denote the value $2\varepsilon(1/2 - 2\varepsilon)^{-1}$ by ρ. Summing the bounds on all the terms $\|x_{I'_j} - x^*_{I'_j}\|_1$ produces

$$\begin{aligned}
\sum_{j \geq 1}\|x_{I'_j} - x^*_{I'_j}\|_1 &\leq (1/2 - 2\varepsilon)^{-1}\Big(\Delta + \sum_{j \geq 1}\sum_{i < j}\Delta_{ij}\Big) \\
&= (1/2 - 2\varepsilon)^{-1}\Big(\Delta + \sum_{i \geq 1}\sum_{j > i}\Delta_{ij}\Big) \\
&\leq (1/2 - 2\varepsilon)^{-1}\Big(\Delta + \frac{\rho}{1 - \rho}\Delta\Big) \quad \text{(By Lemma 7.3.7)} \\
&= (1/2 - 2\varepsilon)^{-1}\Delta(1 - \rho)^{-1} \ .
\end{aligned}$$

Now we can write:

$$\begin{aligned}
\|x - \hat{x}\|_1 &= \|x_R\|_1 + \sum_{j \geq 1}\|x_{I'_j} - x^*_{I'_j}\|_1 \leq \\
&\leq \|x_R\|_1 + \Delta(1/2 - 2\varepsilon)^{-1}(1 - \rho)^{-1} = \\
&= \|x_R\|_1 + \Delta(1/2 - 4\varepsilon)^{-1} \leq \|x_R\|_1\Big(1 + \frac{4\varepsilon}{1 - 8\varepsilon}\Big) + \frac{2}{(1 - 8\varepsilon)d}\|\nu\|_1 \ .
\end{aligned}$$

We finish the proof by plugging in the bound on $\|x_R\|_1$ given by Lemma 7.3.6. $\quad\square$

7.4 Appendix

7.4.1 Space efficient method for computing $\bar{I}$

We will describe how it is possible reduce the storage requirement to $O(kd \log n)$ bits, at the expense of increasing the running time to $O(nd \log n)$. Some constant factors in the parameters have to be increased as well. Suppose (in this

paragraph only) that $\bar{I}$ is redefined so that the constant 2ε in (7.2) is changed to 6ε (any value higher than 3ε would in principle work). Observe that any element of $\bar{I} \setminus I$ that has at least $(1 - 4\varepsilon)d$ unique neighbors within the set $\bar{I} \setminus I$ must have at least $2\varepsilon d$ neighbors shared with the neighbors of I. Therefore at least half of the elements of $\bar{I} \setminus I$ belong to the set

$$T_1 = \{i \in \{1, 2, \ldots, n\} \setminus I \ : \ |\Gamma(i) \cap \Gamma(I)| \geq 2\varepsilon\} \ .$$

The algorithm first finds T_1, which takes $O(nd)$ time. The set $\bar{I}$ is initialized to I, and it will be expanded incrementally. To efficiently determine $T_1 \cap \bar{I}$ the algorithm constructs a priority queue over the set T_1 with the priority of element i being $|\Gamma(i) \cap \Gamma(\bar{I})|$; in this part the process is the same as earlier. Time $O(kd)$ is spent on finding the intersection of T_1 and $\bar{I}$, since T_1 can have at most $|I| = 2k$ elements. It is clear how the algorithm can proceed to run in total time $O(nd \log n)$.

7.4.2 Point queries and heavy hitters

In this section we turn to the streaming model of computation, and look at the related problems of answering point queries and reporting a set of heavy hitters in a stream. We are interested in having reasonably good deterministic structures for these problems. The vector x is now implicitly defined by a stream of update operations. In the streaming context it is assumed that noise ν is zero.

For point queries, the same kind of linear sketch that we used earlier, together with Step 1 of the recovery algorithm, immediately yields a structure with a good *average* error. In general, it is possible to provide meaningful approximations for only $O(s)$ coordinates of x, those of relatively high magnitudes. Further, since we want to use a fixed matrix A for any possible vector x and also want to have $m = O(s \log n)$, we may not provide a good approximation guarantee for *every* "heavy" entry of x. Yet, we may provide approximations with a reasonably small average error, as stated in Theorem 7.3.3. To get a clear bound in the streaming context, we put $M = \emptyset$ and $\alpha = 0$ in Theorem 7.3.3. We may set a fixed value to ε, say $\varepsilon = 1/12$. Then on any index set I of size exactly $s/2$ the average error of point estimates is at most $\frac{1}{s}\|x\|_1$. The space requirements for the data structure come mainly from the vector c. Assuming an explicit optimal expander G, the memory space spent on the description of G may be neglected (it does not depend on s). The size of the sketch vector c would be $O(s \log n \log M)$ bits, where M is an upper bound on $\|x\|_1$. It seems slightly better to tune the accuracy of the estimates by scaling the parameter s, rather than ε, when suitable. In that case, the trade-off between the space usage and the average error is also clearer. Yet, if we want to guarantee good point estimates (on average) on index sets of sizes considerably smaller than s, then taking a smaller value for parameter ε is necessary. Our deterministic structure (yet, still non-explicit) uses the same amount of space (up to constant factor) as the randomized Count-Min sketch [CM05] when both structures give the same value of the error bound, but the type of our error bound is somewhat different.

Let $HH_\phi^p(x) = \{i \ : \ |x_i|^p \geq \phi\|x\|_p^p\}$ be the set of heavy hitters of the vector x. It seems hard (if possible) in the general streaming model to report all the heavy

hitters using space $o(1/\phi^2)$. Here we will investigate the case $x \geq 0$, which often appears in applications. Some streaming algorithms work only under the non-negativity assumption. This model stands between the most general streaming model and the insertions-only model. We solve a version of the problem that has been studied in the literature, showing that there exists a deterministic sketching structure of size $O(1/\phi \cdot \log m \log M)$ bits that reports a set $S \subset \{1, 2, \ldots, n\}$ such that $|S| = O(1/\phi)$ and S contains all ϕ-heavy hitters of x in the ℓ_1 norm, i.e., $HH_\phi^1(x) \subset S$. We will give two simple solutions.

A common way of reporting heavy hitters is via a structure for reporting point queries. We get one solution by using our structure for point queries. We may set $s = 2/\phi$, $\varepsilon = 1/12$, and choose to return the set of indexes $S = \{i : |x_i^*| \geq \phi\|x\|_1\}$. Because of the assumption that $x \geq 0$ we can easily maintain the value $\|x\|_1$ during updates of x. We establish a simple upper bound of $|S| < 3/\phi$. According to Corollary 7.3.5 there will be less than $1/\phi$ elements from the set $\{i : |x_i| < \frac{\phi}{2}\|x\|_1\}$ whose estimates x_i^* are in the range $x_i^* \geq \phi\|x\|_1$, and as such they might appear to be of higher magnitude than some of the real ϕ-heavy hitters. To that number we added $2/\phi$, which is the upper bound on the number of elements from $\{i : |x_i| \geq \frac{\phi}{2}\|x\|_1\}$. The constant factors in this approach can be slightly optimized, but we can achieve considerably lower constants with a bit different method. This second method is a more direct solution to the problem of finding heavy hitters.

We again use a linear sketch of form $c = Ax$, where A is an $m \times n$ adjacency matrix of an expander G, but this time with parameters $((1-\varepsilon)\phi)^{-1} + 1$, d, ε). The algorithm determines the set $J = \{j : c_j \geq \phi\|x\|_1\}$. Since $\|c\|_1 = d\|x\|_1$, it follows that $|J| \leq d/\phi$. The algorithm outputs the set $S = \{i : \Gamma(i) \subset J\}$. Clearly, all the ϕ-heavy hitters are contained in S. It holds that $|S| \leq ((1-\varepsilon)\phi)^{-1}$, as otherwise the expansion property would be violated.

Appendix A

An Expander Conjecture

In this brief chapter we describe a family of efficiently constructible bipartite graphs with highly unbalanced sides. We conjecture that members of this family are expander graphs, and that expansion arbitrarily close to the degree can be achieved. The construction is algebraic and simple, yet a potential proof is unlikely to be simple.

For a vector space $\mathbb{F}^q$, let $\langle \cdot, \cdot \rangle : \mathbb{F}^q \times \mathbb{F}^q \to \mathbb{F}$ be the symmetric bilinear form given by $\langle x, y \rangle = \sum_{i=1}^{q} x_i y_i$. Let $\rho_n(i, j, y)$ be the vector from $\mathbb{F}^{nq}$ that at coordinate $(i-1)q + k$ has value y_k, $1 \le k \le q$, at coordinate $(j-1)q + k$ has value $-y_k$, and all other coordinates equal to zero. The left degree of the graph is denoted by d. The maximum size of subsets for which expansion should hold is denoted by N.

We identify the left vertex set with vector space $\mathbb{F}^q$, where $\mathbb{F}$ is a finite field, and the right vertex set is identified with $[d] \times \mathbb{F}$. Let $(a_1, a_2, \ldots, a_d)$ be a sequence of elements from $\mathbb{F}^q$. For any $x \in \mathbb{F}^q$ and $i \in [d]$, the ith edge incident from x is incident to the element $(i, \langle x, a_i \rangle)$. Let A denote the set $\{a_1, a_2, \ldots, a_d\}$. For such a graph to indeed be an expander, the field $\mathbb{F}$ and the elements of A need to satisfy some conditions. As an example of inappropriate setup take $q = 2$ and $a_i = (1, i)$, $1 \le i \le d$, with $\mathbb{F}$ being a prime field. It is easy to see that the resulting graph has a poor expansion. Examples of similar kind can as well be created for non-prime fields, including fields of small characteristic.

The value of the left degree is expressed as $d = \frac{1}{\varepsilon} cq \log N$, where $c \ge 2$ is a constant value. Any q-element subset of A should be linearly independent (the condition could be relaxed by requiring that any $O(q)$-element subset of A has full rank in $\mathbb{F}^q$). This is a natural requirement on the set A, but it is not sufficient. It is still unclear what additional property may be required. The size of $\mathbb{F}$ should be at least $\Omega(\frac{1}{\varepsilon} N)$. Providing a full characterization of the parameter values, including the set A, that give good expansion seems exceedingly difficult. A more reasonable goal would be to prove the conjecture with a sufficient set of conditions. Perhaps choosing A uniformly at random works with a high probability.

Let S be a subset of $\mathbb{F}^q$ of size n. Denote the elements of S by x_i, $1 \le i \le n$. If the kth neighbour of x_i is shared with some neighbour of x_j then that has to be exactly the kth neighbour of x_j; this was the reason for having $[d] \times \mathbb{F}$ as the right vertex set. Each collision between a pair of elements from S corresponds

to an equality of type $\langle x_i, a_k \rangle = \langle x_j, a_k \rangle$. Based on those equalities we construct a big matrix with nq columns. Each equality in the system is either directly represented by one row in the matrix or can be easily deducted from a set of rows (we do not want to include rows that are "obviously" linearly dependent on the remaining ones). Concretely, for any chain of equalities $\langle x_{i_1}, a_k \rangle = \langle x_{i_2}, a_k \rangle = \cdots = \langle x_{i_m}, a_k \rangle$, where $i_1 < i_2 < \ldots < i_m$ and $\langle x_j, a_k \rangle \neq \langle x_{i_1}, a_k \rangle$ for any $j \notin \{i_1, i_2, \ldots, i_m\}$, we include the following vectors as rows of the matrix: $\rho_n(i_1, i_2, a_k), \rho_n(i_2, i_3, a_k), \ldots, \rho_n(i_{m-1}, i_m, a_k)$. Through this construction we end up with a matrix having $dn - |\Gamma(S)|$ rows (because for any right vertex $v \in \Gamma(S)$ there are $|\Gamma(v) \cap S| - 1$ matrix rows that are related to v). Denote this matrix by M. The null space of M includes a vector that is a representation of the set S, that is the vector $((x_1)_1, (x_1)_2, \ldots, (x_1)_q, (x_2)_1, (x_2)_2, \ldots, (x_n)_q)$. Observe that $\mathrm{Null}(M)$ also includes all vectors z such that $z_i = z_{i+q}$, for every $i \leq (n-1)q$ (i.e. any vector produced by "replication" of a vector from $\mathbb{F}^q$). Hence, the dimension of the null space of M is at least q. If we could prove that $\mathrm{rank}(M) = (n-1)q$ when $|\Gamma(S)|$ is not large enough and $n \leq N$, that would imply that the null space of M consists only of trivial solutions, meaning that the vector representation of S does not belong to $\mathrm{Null}(M)$, which is a contradiction.

To be precise about the threshold for $|\Gamma(S)|$, we consider the expansion property violated when

$$|\Gamma(S)| \leq n(d - cq \log n) \ ,$$

assuming that $n \leq N$. The value on the right hand size is somewhat higher than $(1-\varepsilon)dn$, unless n is very close to N. From the upper bound on $|\Gamma(S)|$ it follows that M has at least $n \cdot cq \log n$ rows. We may suppose that the given set S is a minimal set that violates the expansion property, meaning it has no subset S' of size $m < n$ such that $|\Gamma(S')| \leq m(d - cq \log m)$. We may naturally construct a *collision graph* having n vertices and $dn - |\Gamma(S)|$ edges. The minimality condition for S ensures that no vertex subset has a small edge cut with the rest of that graph. Further, the minimum degree of the collision graph is at least $cq \log n$.

We have tried various proof methods. The methods can be divided in two general approaches: analyzing linear combinations of the rows of M, and computing the determinant of various matrices derived from M. Some restrictions, such as requiring the field $\mathbb{F}$ to be of characteristic two, were also tried. Despite all the efforts, a proof remains elusive. We have not considered spectral methods because we really wanted the proof to work for small ε as well. Besides, it would be interesting to have a self-contained proof of expansion that does not rely on spectral methods at any level. We still believe that this kind of graphs is worth looking into. The succinctness of the graph representation and the speed of evaluating the neighbour function are very appealing.

Bibliography

[ABKU99] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

[AFGV97] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On sorting strings in external memory. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 540–548. ACM, 1997.

[AH92] Susanne Albers and Torben Hagerup. Improved parallel integer sorting without concurrent writing. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 463–472, 1992.

[AHNR98] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *J. Comput. Syst. Sci.*, 57(1):74–93, 1998.

[AMRT96] Arne Andersson, Peter Bro Miltersen, Søren Riis, and Mikkel Thorup. Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 441–450. IEEE Computer Society, 1996.

[AN96] Noga Alon and Moni Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16(4-5):434–449, 1996.

[And95] Arne Andersson. Sublogarithmic searching without multiplications. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 655–663. IEEE Computer Society, 1995.

[And96] Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 135–141. IEEE Computer Society, 1996.

[AT00] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 335–342. ACM press, 2000.

[AV88]   Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[BCSV00]   Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: the heavily loaded case. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 745–754. ACM press, 2000.

[BF02]   Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002.

[BF03]   Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC)*, pages 307–315. ACM, 2003.

[BF06]   Gerth Stølting Brodal and Rolf Fagerberg. Cache-oblivious string dictionaries. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 581–590, 2006.

[BFCK06]   Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string B-trees. In *Proceedings of the 25th ACM Symposium on Principles of Database Systems (PODS)*, pages 233–242. ACM, 2006.

[BG07]   Guy E. Blelloch and Daniel Golovin. Strongly history-independent hashing with applications. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 272–282, 2007.

[BGI+08]   Radu Berinde, Anna C. Gilbert, Piotr Indyk, Howard J. Karloff, and Martin J. Strauss. Combining geometry and combinatorics: A unified approach to sparse signal recovery. *CoRR*, 2008. Preprint.

[BHT05]   Mette Berger, Esben Rune Hansen, and Peter Tiedemann. Expander based dictionary data structures. Master's thesis, IT-University of Copenhagen, 2005.

[BIP09]   Khanh Do Ba, Piotr Indyk, and Eric Price. Lower bounds for sparse recovery. *Preprint*, 2009.

[BIR08]   Radu Berinde, Piotr Indyk, and Milan Ružić. Practical near-optimal sparse recovery in the l_1 norm. In *46th Annual Allerton Conference on Communication, Control, and Computing*, 2008.

[BMM97]   Andrej Brodnik, Peter Bro Miltersen, and J. Ian Munro. Trans-dichotomous algorithms without multiplication - some upper and lower bounds. In *Proceedings of the 5th Int. Workshop on Algorithms and Data Structures*, volume 1272 of *Lecture Notes in Computer Science*, pages 426–439. Springer, 1997.

[BMQ98]   John R. Black, Charles U. Martel, and Hongbin Qi. Graph and hashing algorithms for modern architectures: Design and performance. In *Proceedings of the 2nd International Workshop on Algorithm Engineering (WAE)*, pages 37–48. Max-Planck-Institut für Informatik, 1998.

[BMRV02]   Harry Buhrman, Peter Bro Miltersen, Jaikumar Radhakrishnan, and Srinivasan Venkatesh. Are bitvectors optimal? *SIAM J. Comput.*, 31(6):1723–1744, 2002.

[CCFC04]   Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.

[CDD06]   A. Cohen, W. Dahmen, and R. DeVore. Compressed sensing and best k-term approximation. *Preprint*, 2006.

[Che52]   Herman Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23(4):493–507, 1952.

[CM05]   Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[CM06]   Graham Cormode and S. Muthukrishnan. Combinatorial algorithms for compressed sensing. In *SIROCCO*, volume 4056 of *Lecture Notes in Computer Science*, pages 280–294. Springer, 2006.

[CRT06a]   E. Candès, J. Romberg, and T. Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. *IEEE Inf. Theory*, 52(2):489–509, 2006.

[CRT06b]   E. J. Candès, J. Romberg, and T. Tao. Stable signal recovery from incomplete and inaccurate measurements. *Comm. Pure Appl. Math.*, 59(8):1208–1223, 2006.

[CRVW02]   Michael R. Capalbo, Omer Reingold, Salil P. Vadhan, and Avi Wigderson. Randomness conductors and constant-degree lossless expanders. In *Proceedings of the 34nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 659–668, 2002.

[CW79]   Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

[DDT+08]   M. Duarte, M. Davenport, D. Takhar, J. Laska, T. Sun, K. Kelly, and R. Baraniuk. Single-pixel imaging via compressive sampling. *IEEE Signal Processing Magazine*, 2008.

[DG77]   Persi Diaconis and R. L. Graham. Spearman's footrule as a measure of disarray. *J. Royal Statistical Society*, 39:262–268, 1977.

[DGMP92]   Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proceedings of the 19th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer-Verlag, 1992.

[DHKP97]   Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.

[Die89]   Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proceedings of the Workshop on Algorithms and Data Structures (WADS '89)*, volume 382 of *Lecture Notes in Computer Science*, pages 39–46. Springer-Verlag, 1989.

[Die96]   Martin Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1046 of *Lecture Notes in Computer Science*, pages 569–580. Springer-Verlag, 1996.

[DKM+94]   Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Computing*, 23(4):738–761, 1994.

[DM08]   Wei Dai and Olgica Milenkovic. Subspace pursuit for compressive sensing: Closing the gap between performance and complexity. *CoRR*, abs/0803.0811, 2008. Preprint.

[DMT07]   Cynthia Dwork, Frank McSherry, and Kunal Talwar. The price of privacy and the limits of lp decoding. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC)*, pages 85–94, 2007.

[Don06]   D. L. Donoho. Compressed Sensing. *IEEE Trans. Info. Theory*, 52(4):1289–1306, 2006.

[DW05]   Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, pages 166–178. Springer, 2005.

[DWB05]   Marco Duarte, Michael Wakin, and Richard Baraniuk. Fast reconstruction of piecewise smooth signals from random projections. In *Proceedings of SPARS '05*, 2005.

[FG99]   Paolo Ferragina and Roberto Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[FKS84]   Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

[FLPR99]   Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 285–298, 1999.

[FPP06]   Rolf Fagerberg, Anna Pagh, and Rasmus Pagh. External string sorting: Faster and cache-oblivious. In *Proceedings of the 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 3884 of *Lecture Notes in Computer Science*, pages 68–79. Springer, 2006.

[FS89]   Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC)*, pages 345–354. ACM, 1989.

[FW93]   Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[FW94]   Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3), 1994.

[GG81]   Ofer Gabber and Zvi Galil. Explicit constructions of linear-sized superconcentrators. *J. Comput. Syst. Sci.*, 22(3):407–420, 1981.

[GLR08]   Venkatesan Guruswami, James R. Lee, and Alexander A. Razborov. Almost euclidean subspaces of l^n_1 via expander codes. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 353–362, 2008.

[Gon81]   Gaston H. Gonnet. Expected length of the longest probe sequence in hash code searching. *J. ACM*, 28(2):289–304, 1981.

[Gro06]   Rice DSP Group. Compressed sensing resources. *Available at* `http://www.dsp.ece.rice.edu/cs/`, 2006.

[GSTV06]   A. C. Gilbert, M. J. Strauss, J. A. Tropp, and R. Vershynin. Algorithmic linear dimension reduction in the ℓ_1 norm for sparse vectors. Submitted for publication, 2006.

[GSTV07]   Anna C. Gilbert, Martin J. Strauss, Joel A. Tropp, and Roman Vershynin. One sketch for all: fast algorithms for compressed sensing. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC)*, pages 237–246, 2007.

[GUV07]   Venkatesan Guruswami, Christopher Umans, and Salil P. Vadhan. Unbalanced expanders and randomness extractors from Parvaresh-Vardy codes. In *22nd Annual IEEE Conference on Computational Complexity (CCC)*, pages 96–108, 2007.

[Hag98a]   Torben Hagerup. Simpler and faster dictionaries on the AC^0 RAM. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 79–90. Springer, 1998.

[Hag98b]   Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer-Verlag, 1998.

[Han04]   Yijie Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. *J. Algorithms*, 50(1):96–105, 2004.

[HL05]   Gregory L. Heileman and Wenbin Luo. How caching affects hashing. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 141–154. SIAM, 2005.

[HLW06]   Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bull. Amer. Math. Soc.*, 43(4):439–561, 2006.

[HMP01]   Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. *J. Algorithms*, 41(1):69–85, 2001.

[Hoe63]   Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[Ind07]   Piotr Indyk. Sketching, streaming and sublinear-space algorithms. *Graduate course notes, available at* `http://stellar.mit.edu/S/course/6/fa07/6.895/`, 2007.

[Ind08]   Piotr Indyk. Explicit constructions for compressed sensing of sparse signals. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 30–33, 2008.

[IR08]   Piotr Indyk and Milan Ružić. Near-optimal sparse recovery in the l_1 norm. In *49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 199–207. IEEE, 2008.

[Knu63]   Donald E. Knuth. Notes on "open" addressing, 1963. Unpublished memorandum. Available at `http://citeseer.ist.psu.edu/knuth63notes.html`.

[Knu73]   Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley Publishing Co., Reading., Mass., 1973.

[Knu98]   Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Co., Reading, Mass., second edition, 1998.

[KR93]   Howard J. Karloff and Prabhakar Raghavan. Randomized algorithms and pseudorandom numbers. *J. ACM*, 40(3):454–476, 1993.

[KRS90]   C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95–132, 1990.

[KS03]   Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003.

[KT07]   B. S. Kashin and V. N. Temlyakov. A remark on compressed sensing. *Preprint*, 2007.

[Mar73]   G. Margulis. Explicit constructions of expanders. *Problemy Peredači Informacii*, 9:71–80, 1973.

[Meh84]   Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer, 1984.

[Mil98]   Peter Bro Miltersen. Error correcting codes, perfect hashing circuits, and deterministic dynamic dictionaries. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 556–563. ACM Press, 1998.

[Mut03]   S. Muthukrishnan. Data streams: Algorithms and applications (invited talk at soda'03). 2003. Available at http://athos.rutgers.edu/∼muthu/stream-1-1.ps.

[MV84]   Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Inf.*, 21:339–374, 1984.

[NT09]   D. Needell and J. A. Tropp. CoSaMP: Iterative signal recovery from incomplete and inaccurate samples. *Appl. Comp. Harmonic Anal.*, 26:301–321, 2009.

[NV09]   D. Needell and R. Vershynin. Uniform uncertainty principle and signal recovery via regularized orthogonal matching pursuit. *Foundations of Computational Mathematics*, pages 317–334, 2009.

[ÖP02]   Anna Östlin and Rasmus Pagh. One-probe search. In *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 2380 of *Lecture Notes in Computer Science*, pages 439–450. Springer, 2002.

[OvL81]   Mark H. Overmars and Jan van Leeuwen. Worst-case optimal in-
           sertion and deletion methods for decomposable searching problems.
           Inf. Proc. Lett., 12(4):168–173, 1981.

[Pag00]   Rasmus Pagh. A trade-off for worst-case efficient dictionaries.
           Nordic J. Comput., 7(3):151–163, 2000.

[Pe98]   H. Prodinger and W. Szpankowski (eds.). Special issue on average
           case analysis of algorithms. *Algorithmica*, 22(4), 1998. Preface.

[PR04]   Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J.
           Algorithms*, 51(2):122–144, 2004.

[PT06]   Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for pre-
           decessor search. In *Proceedings of the 38th Annual ACM Symposium
           on Theory of Computing (STOC)*, pages 232–240. ACM, 2006.

[Ram96]   Rajeev Raman. Priority queues: Small, monotone and trans-
           dichotomous. In *Proceedings of the 4th Annual European Symposium
           on Algorithms (ESA)*, volume 1136 of *Lecture Notes in Computer
           Science*, pages 121–137. Springer-Verlag, 1996.

[Ruž07]   Milan Ružić. Making deterministic signatures quickly. In *Proceed-
           ings of the 18th ACM-SIAM Symposium on Discrete Algorithms
           (SODA)*, pages 900–909. ACM and SIAM, 2007.

[Ruž08a]   Milan Ružić. Constructing efficient dictionaries in close to sort-
           ing time. In *Proceedings of the 35th International Colloquium on
           Automata, Languages and Programming (ICALP)*, pages 84–95.
           Springer, 2008.

[Ruž08b]   Milan Ružić. Uniform deterministic dictionaries. *ACM Transactions
           on Algorithms*, 4(1):Article 1, 2008.

[Ruž09]   Milan Ružić. Making deterministic signatures quickly. *ACM Trans-
           actions on Algorithms*, 5(3):Article 26, 2009.

[Sie04]   Alan Siegel. On universal classes of extremely random constant-time
           hash functions. *SIAM J. Comput.*, 33(3):505–543, 2004.

[SS90]   Jeanette P. Schmidt and Alan Siegel. The analysis of closed hashing
           under limited randomness (extended abstract). In *Proceedings of the
           22nd Annual ACM Symposium on Theory of Computing (STOC)*,
           pages 224–234. ACM, 1990.

[SS95]   Alan Siegel and Jeanette P. Schmidt. Closed hashing is computable
           and optimally randomizable with universal hash functions. Techni-
           cal Report TR1995-687, New York University, 1995.

[SSS95]   Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-
           Hoeffding bounds for applications with limited independence. *SIAM
           J. Discrete Math.*, 8(2):223–250, 1995.

[TG05]   Joel A. Tropp and Anna C. Gilbert. Signal recovery from partial information via Orthogonal Matching Pursuit. Submitted for publication, 2005.

[Tho03]   Mikkel Thorup. On AC^0 implementations of fusion trees and atomic heaps. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 699–707, 2003.

[Tho09]   Mikkel Thorup. String hashing for linear probing. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 655–664. SIAM, 2009.

[TLW$^+$06] D. Takhar, J. Laska, M. B. Wakin, M. F. Duarte, D. Baron, S. Sarvotham, K. Kelly, and R. G. Baraniuk. A new compressive imaging camera architecture using optical-domain compression. In *Proc. IS&T/SPIE Symposium on Electronic Imaging*, 2006.

[Tro04]   Joel A. Tropp. Greed is good: Algorithmic results for sparse approximation. *IEEE Trans. Inform. Theory*, 50(10):2231–2242, 2004.

[TS02]   Amnon Ta-Shma. Storing information with extractors. *Inf. Process. Lett.*, 83(5):267–274, 2002.

[TSUZ01]   Amnon Ta-Shma, Christopher Umans, and David Zuckerman. Lossless condensers, unbalanced expanders, and extractors. In *Proceedings of the 33nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 143–152, 2001.

[TY79]   Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Commun. ACM*, 22(11):606–611, 1979.

[TZ04]   Mikkel Thorup and Yin Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 615–624, 2004.

[vEBKZ77] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[VS94]   Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory i: Two-level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[WC81]   Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981.

[XH07]   W. Xu and B. Hassibi. Efficient compressive sensing with determinstic guarantees using expander graphs. *IEEE Information Theory Workshop*, 2007.