Alcom-FT Technical Report Series
ALCOMFT-TR-01-126

Lossy Dictionaries

Rasmus Pagh* and Flemming Friche Rodler

BRICS**
Department of Computer Science
University of Aarhus, Denmark
{pagh,ffr}@brics.dk

Abstract. Bloom filtering is an important technique for space efficient storage of a conservative approximation of a set S. The set stored may have up to some specified number of "false positive" members, but all elements of S are included. In this paper we consider "lossy dictionaries" that are allowed to also have false negatives. This relaxation allows a very fast and simple data structure making almost optimal use of memory. Being more time efficient than Bloom filters, we believe our data structure to be well suited for replacing Bloom filters in some applications. Also, the fact that our data structure supports information associated to keys paves the way for new applications, as illustrated by an application in lossy image compression.

1 Introduction

Dictionaries are part of many algorithms and data structures. A dictionary provides access to information indexed by a set S of *keys*: Given a key, it returns the associated information or reports that the key is not in the set. In this paper we will not be concerned with updates, i.e., we consider the *static* dictionary problem. The main parameters of interest are of course the space used by the dictionary and the time for looking up information. We will assume keys as well as the information associated with keys to have a fixed size.

A large literature has grown around the problem of constructing efficient dictionaries, and theoretically satisfying solutions have been found. Often a slightly easier problem has been considered, namely the *membership* problem, which is the dictionary problem without associated information. It is usually easy to derive a dictionary from a solution to the membership problem, using just extra space corresponding to the associated information. In this paper, we are particularly interested in dictionary and membership schemes using little memory. Let n denote the size of the key set S. It has been shown that when keys are w-bit machine words, lookups can be performed in constant time in a membership data structure occupying $B + o(B)$ bits of memory, where $B = \log_2 \binom{2^w}{n}$ is the

* Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).
** Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

minimum amount of memory needed to be able to represent any subset of size n [2]. However, constant factors in the lower order term and lookup time makes this and similar schemes less than one could hope for from an applied point of view. Also, difficulty of implementation is an obstacle to practical use. In total, current schemes with asymptotically optimal space usage appear to be mainly of theoretical interest.

If one relaxes the requirements to the membership data structure, allowing it to store a slightly different key set than intended, new possibilities arise. A technique finding many applications in practice is *Bloom filtering* [1]. This technique allows space-efficient storage of a superset S' of the key set S, such that $S' \cap \complement S$ is no more than an ϵ fraction of $\complement S$, the complement of S. About $\log_2(1/\epsilon)$ bits per key in S are necessary and sufficient for this [5], thus giving significant savings compared to a dictionary using $B \approx n \log_2(\frac{eu}{n})$ bits, where u denotes the size of the set of possible keys. Lookup of a key using Bloom filtering requires $O(\log(1/\epsilon))$ memory accesses and is thus relatively slow compared to other hashing schemes when ϵ is small. Bloom filtering has applications in, for example, cooperative caching and differential files, where one wants no more than a small chance that an expensive lookup is done in vain. Bloom filtering differs from most other hashing techniques in that is does *not* yield a solution to the dictionary problem.

In this paper, we introduce the concept of *lossy dictionaries* that can have false positives (like Bloom filters), but also false negatives. That is, some keys in S (with associated information) are thrown away when constructing the dictionary. For false positives, any associated information may be returned. We let each key in S have a weight and try to maximize the sum of weights of keys in the dictionary under a given space constraint. This problem, with no false positives allowed, arises naturally in lossy image compression, and is potentially interesting for caching applications. Also, a dictionary with two-sided errors could take the place of Bloom filters in cooperative caching. We find that some very simple and efficient algorithms have surprisingly good behavior with respect to keeping the keys of largest weight.

1.1 Related work

Most previous work on static dictionaries has considered the unit cost RAM and the case where keys are machine words of w bits. The first dictionary with worst case constant lookup time using $O(n)$ words of space was constructed by Fredman et al. [8]. For constant $\epsilon > 0$, the space usage is $O(B)$ when $2^w > n^{1+\epsilon}$, but in general the data structure may use space $\Omega(Bw)$. The space usage has been lowered to $B + o(B)$ bits in a dictionary of Brodnik and Munro [2]. The lower order term was subsequently improved to $o(n) + O(\log w)$ bits by Pagh [12].

The membership problem with false positives was first considered by Bloom [1]. Apart from Bloom filtering the paper presents a less space efficient data structure that is readily turned into a lossy dictionary with only false positives. However, the space usage of the derived lossy dictionary is not optimal. Carter et al. [5]

provided a lower bound of $n \log(1/\epsilon)$ bits on the space needed to solve membership with an ϵ fraction false positives, and gave data structures with various lookup time matching or nearly matching this bound. Though none of the membership data structures have constant lookup time, such a data structure follows by plugging the abovementioned results on space optimal dictionaries into a general reduction provided in [5]. In fact, the dictionary of [12] can be easily modified to a lossy dictionary with false positives, thus also supporting associated information, using $O(n + \log w)$ bits more than the lower bound.

Another relaxation of the membership problem was recently considered by Buhrman et al. [3]. They allow the lookup procedure to use randomization and to have some probability of error. For two-sided error ϵ they achieve space $O(nw/\epsilon^2)$ and lookups using just *one* bit probe. To do the same without false negatives it is shown that $O(n^2 w/\epsilon^2)$ bits suffice and that this is essentially optimal. Schemes using more bit probes and less space are also investigated. If one fixes the random bits of the lookup procedure appropriately, the result is a lossy dictionary with error ϵ. However, the preprocessing and lookup steps in [3] are not shown to be doable in polynomial time, so no efficient lossy dictionary follows.

2 Lossy dictionaries

Consider a set S containing keys $k_1, \ldots, k_n$ with associated information $a_1, \ldots, a_n$ and positive weights $w_1, \ldots, w_n$, where $w_i \geq w_{i+1}$ for $i = 1, \ldots, n-1$. Suppose we are given an upper bound m on available space and an error parameter ϵ. The *lossy dictionary problem* for $\epsilon = 0$ is to store a subset of the keys in S and corresponding associated information in a data structure of m bits, trying to optimize the sum of weights of included keys, referred to as the *value*. For general ϵ we allow the dictionary to contain also a fraction ϵ keys from the complement of S.

Our starting point is a static dictionary recently described in [13]. In this dictionary, two hash tables T_1 and T_2 are used together with two hash functions $h_1, h_2 : U \to \{1, \ldots, r/2\}$, where U is the set of all keys and r denotes the combined size of the hash tables, assumed to be even. A key $x \in S$ is stored in either cell $h_1(x)$ of T_1 or cell $h_2(x)$ of T_2. It was shown that if $r \geq (2+\epsilon)\, n$ for some constant $\epsilon > 0$, and h_1, h_2 are chosen independently and uniformly at random from an $O(\log r)$-wise independent family of functions, there exists a way of arranging the keys in the tables according to the hash functions with probability $1 - O(1/r)$. This gives a dictionary utilizing about 50% of the hash table cells. The arrangement of keys was shown to be computable in expected linear time.

Siegel [17] has constructed an $O(\log r)$-wise independent family whose functions can be evaluated in constant time when the domain has size $r^{O(1)}$. The initialization time and space needed can be made n^ϵ for any $\epsilon > 0$ of our choice. We can use it also for larger key domains by first using a function from a universal family [4] to map the set injectively to a range of size $O(r^2)$. Unfortunately, with current knowledge of explicit expander graphs, the constant in the evaluation time is not practical. However, recent empirical work shows that the above scheme works very well in practice with simpler and more efficient hash function families [14]. So even though we will analyze our scheme under the assumption that hash function values are truly random and independent, we believe that algorithms with efficiently implementable hash function classes work more or less equally well. This belief is backed by experiments described in Sect. 3.

2.1 Our data structure

The idea behind our lossy dictionary, compared to the static dictionary described above, is to try to fill the hash tables almost completely, working with key sets of size similar to or larger than r. Each key has two hash table cells to which it can be "matched". Thus, given a pair of hash functions, the problem of finding a maximum value subset of S that can be arranged into the hash tables is a maximum weight matching problem that can be solved in polynomial time, see e.g. [6]. In Sect. 2.2 we will present an algorithm that finds an optimal solution in nearly linear time, exploiting its structural properties. Also, we give a lower bound on the expected value of the optimal solution, for which an empirical evaluation is also given in Sect. 3. However, let us first describe the data structure in more detail, assuming that we know which keys are to be represented in which hash table cells.

The main concept used is that of a *quotient function* q of a hash function h, defined to be a function such that the mapping $k \mapsto (h(k), q(k))$ is injective [12]. When storing a key k in cell $h(k)$ of a hash table, it is sufficient to store $q(k)$ to uniquely identify k among all other elements hashing to $h(k)$. The idea is, of course, that storing $q(k)$ may require fewer bits than storing k itself. If a fraction $O(1/r)$ of all possible keys hashes to each of r hash table cells, there is a quotient function whose values can be stored in $\log u - \log r + O(1)$ bits, where $\log u$ is the number of bits needed to store an arbitrary key. This approach was used in [12] to construct a dictionary using space close to the information theoretical minimum.

As an example, we look at a hash function family from [7] mapping from $\{0,1\}^s$ to $\{0,1\}^t$. It contains functions of the form $h_a(k) = (ak \mod 2^s) \text{ div } 2^{s-t}$ for a odd and $0 \leq a < 2^s$. Letting bit masks and shifts replace modulo and division these hash functions can be evaluated very efficiently. A corresponding family of quotient functions is given by $q_a(k) = (ak \mod 2^s) \bmod 2^{s-t}$. As the domain has size $u = 2^s$ and the range of h_a size $r = 2^t$, function values of q_a can be stored in $\log u - \log r$ bits.

In the following we will assume that corresponding to h_1 and h_2 there are efficiently evaluable quotient functions q_1 and q_2 with a range of $\log u - \log r + O(1)$ bits. For $\epsilon = 0$, we simply store quotient function values in the hash tables, which thus use $r(\log u - \log r + O(1))$ bits, plus the space used for associated information. This is within $O(r)$ bits of the information theoretical minimum for storing r keys and associated information. Given the desired maximum space usage m, one can choose a suitable size r for the tables.

For general ϵ we store $\log(1/\epsilon) + O(1)$ bits of the quotient function value, such that no more than an ϵ fraction of the elements hashing to a given cell agree on the part of the quotient function value stored. This gives at most an ϵ fraction false positives. The space usage is within $O(r)$ bits of optimum for the case where there are no false negatives.

2.2 Analysis and construction algorithm

First note that it may be assumed without loss of generality that weights are distinct. If there are consecutive equal weights $w_j = \cdots = w_k$, we can imagine making them distinct by adding positive integer multiples of some quantity δ much smaller than the difference between any pair of achievable values. For sufficiently small δ, the relative ordering of the values of any two solutions with distinct values will not change. When weights are distinct, the set of keys in an optimal solution is unique, as shown in the following lemma:

Lemma 1. *Suppose that weights are distinct. For $1 \leq i \leq n$, an optimal solution includes key k_i if and only if there is an optimal solution for the set $\{k_1, \ldots, k_{i-1}\}$ such that cell $h_1(k_i)$ of T_1 or cell $h_2(k_i)$ of T_2 is empty. In particular, the set of keys included in an optimal solution is unique.*

Proof. We proceed by induction on n. For $n = 1$ the claim is obvious. In general, the claim follows for $i < n$ by using the induction hypothesis on the set $\{k_1, \ldots, k_i\}$. For $i = n$ consider the unique set K of keys included in an optimal solution for $\{k_1, \ldots, k_{n-1}\}$. If there is an arrangement of K not occupying both cell $h_1(k_n)$ of T_1 and cell $h_2(k_n)$ of T_2, we may add the key k_n to the arrangement which must yield an optimal arrangement. On the other hand, if all arrangements of K occupy both both cell $h_1(k_n)$ of T_1 and cell $h_2(k_n)$ of T_2, there is no way of including k_n without discarding a key in K. However, this would yield a lower value and hence cannot be optimal. □

Given hash functions h_1 and h_2 and a key set K, we define the bipartite graph $G(K)$ with vertex set $\{1, 2\} \times \{1, \ldots, r/2\}$, corresponding in a natural way to hash table cells, and the multiset of edges $\{\{(1, h_1(k)), (2, h_2(k))\} \mid k \in K\}$, corresponding to keys. Note that there may be parallel edges if several keys have the same pair of hash function values. We will use the terms keys/edges and cells/vertices synonymously. Define a connected component of $G(K)$ to be *saturated* if the number of edges is greater than or equal to the number of vertices, i.e., if it is not a tree. We have the following characterization of the key set of the optimal solution:

Lemma 2. *Suppose that weights are distinct. The optimal solution includes key k_i if and only if one of $(1, h_1(k_i))$ and $(2, h_2(k_i))$ is in a non-saturated connected component of $G(\{k_1, \ldots, k_{i-1}\})$.*

Proof. By Lemma 1 it is enough to show that the keys included in an optimal solution for a set of keys K can be arranged such that cell z is empty if and only if the connected component of z in $G(K)$ is not saturated. Consider the key set K' in an optimal solution for K. For every subset $H \subseteq K'$ it must hold that $|h_1(H)| + |h_2(H)| \geq |H|$ since otherwise not all keys could have been placed. Thus, a connected component with key set C is saturated if and only if $|h_1(C \cap K')| + |h_2(C \cap K')| = |C \cap K'|$. In particular, when arranging the keys of $C \cap K'$, where C is the set of keys of a saturated component, every cell in the connected component must be occupied. On the other hand, suppose there is no arrangement of K' such that z, say cell number i of T_1, is empty. Hall's theorem says that there must be a set $H \subseteq K'$ such that $|h_1(H) \backslash \{i\}| + |h_2(H)| < |H|$. In fact, as no other connected components are affected by blocking z, we may chose H as a subset of the keys in the connected component of z. But then the graph of edges in H must contain a cycle, meaning that the connected component is saturated. The case of z being in T_2 is symmetrical. □

The lemma implies that the following algorithm finds the key set S' of the optimal solution, given keys sorted according to decreasing weight:

1. Initialize a union-find data structure for the cells of the hash tables.
2. For each class set a "saturated" flag to `false`.
3. For $i = 1, \ldots, n$:
   (a) Find the equivalence class c_b of cell $h_b(k_i)$ in T_b, for $b = 1, 2$.
   (b) If c_1 or c_2 is not saturated:
      i. Include k_i in the solution.
      ii. Join c_1 and c_2 to a new class c.
      iii. Set the saturated flag of c if it was set for c_1 or c_2.

Operations in a union-find data structure can be implemented quite simply to run in $O(\log^* n)$ amortized time, see e.g. [16].

What remains is arranging the optimal key set S' in the tables. Consider a vertex in $G(S')$ of degree one. It is clear that there must be an arrangement such that the corresponding cell contains the key of the incident edge. Thus, one can iteratively handle edges incident to vertices of degree one and delete them. As we remove the same number of edges and vertices from each connected component, the remaining graph consists of connected components with no more edges than vertices and no vertices of degree one, i.e., cycles. The arrangement of edges in a cycle follows as soon as one key has been put (arbitrarily) into one of the tables. The above steps are easily implemented to run in linear time. Hence we have:

Theorem 3. *For hash functions h_1 and h_2, given a sequence of n keys sorted according to weight, and corresponding hash function values under h_1 and h_2, an optimal solution under h_1 and h_2 can be computed in $O(n \log^* n)$ time.*

We now turn to the problem of estimating the quality of the solution. Again we will use the fact that weights can be assumed to be unique. A consequence of Lemma 1 is that the set of keys in an optimal solution does not depend on the actual weights, but only on the sequence of hash function values. Thus, the expected value of the optimal solution is:

$$\sum_{i=1}^{n} w_i p_{r,i}$$

where $p_{r,i}$ is the probability that the ith key is included in the optimal set of keys. By Lemma 1 and the mentioned result in [13], for constant $\epsilon > 0$ the heaviest $(1-\epsilon)\,r/2$ keys are all included in the optimal solution with probability $1 - O(1/r)$. In particular, $p_{r,i} = 1 - O(1/r)$ for $i \leq (1-\epsilon)\,r/2$. For larger i we derive a lower bound on $p_{r,i}$ as follows. If one of the vertices $(1, h_1(k_i))$ and $(2, h_2(k_i))$ in $G(\{k_1, \ldots, k_{i-1}\})$ is isolated, it follows by Lemma 2 that k_i is in the optimal solution. Under the assumption that hash function values are truly random, $G(\{k_1, \ldots, k_{i-1}\})$ has $i-1$ randomly and independently chosen edges. Thus, we have the bound $p_{r,i} \geq 1-(1-(1-2/r)^{i-1}))^2 \approx 1-(1-e^{-2i/r})^2$. This bound is somewhat pessimistic, as we shall see in Sect. 3.

2.3 Using more tables

In this section we look at a generalization of the two table scheme to schemes with more tables. Unfortunately the algorithm described in Sect. 2.2 does not seem to generalize to more than two tables. An optimal solution can again be found using maximum weight matching, but the time complexity of this solution is not attractive. Instead we can use a variant of the "cuckoo" scheme described in [14]. For two tables it works as follows: If key k_i is the next element to be inserted we store it in cell $h_1(k_i)$ of T_1 pushing the previous occupant, if any, away and thus making it "nestless". If $h_1(k_i)$ was free we are done, otherwise we insert the now "nestless" element in T_2, possibly pushing out another element. This continues until we either find a free cell or loop around unable to find a free cell, in which case k_i is discarded. It follows from the analysis in Sect. 2.2 that this algorithm finds an optimal solution, though not as efficiently as the algorithm given there. When using three or more tables it is not obvious in which of the tables to attempt placing the "nestless" key. One heuristic that works well, as we shall see in the next section, is to simply pick one at random. It is interesting to compare this heuristic to a random walk on a expander graph, which will provably cross any large subset of the vertices with high probability.

3 Experiments

An important performance parameter of our lossy dictionaries is the ability to store many keys with high weight. Our first experiment tests this ability for lossy dictionaries using two and three tables. For comparison, we also test the simple one table scheme that stores in each cell the key of greatest weight hashing to it. The tests were done using truly random hash function values, obtained from a high quality collection of random bits freely available on the internet [11]. Figure 1 shows experimentally determined values of $p_{r,\alpha r}$, the probability that

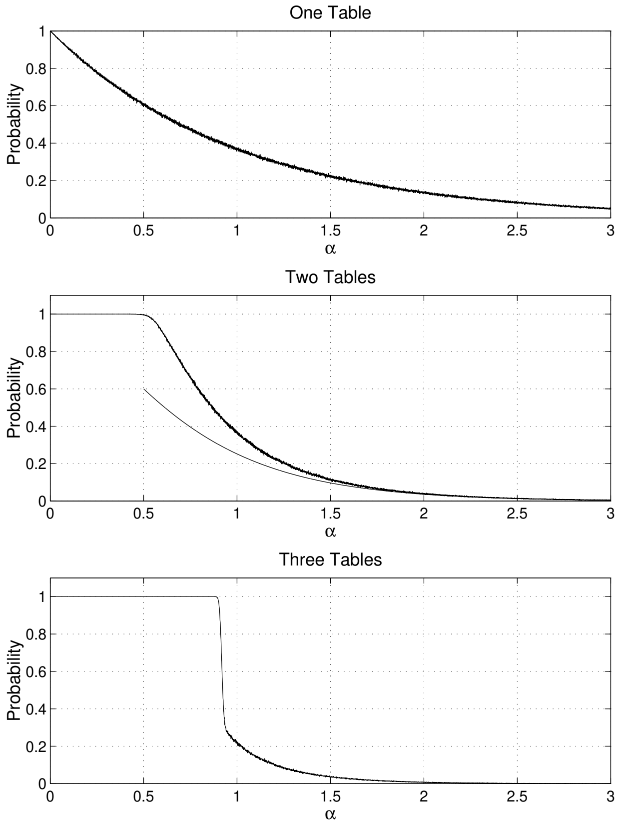

Fig. 1: The probability that the element with (αr)th highest weight is stored. For two tables our lower bound is included.

key with index $i = \alpha r$ is stored in the dictionary, determined from 10^4 trials. For the experiments with one and two tables we used table size $r = 2048$ while for the experiment with three tables we used $r = 1536$. We also tried various other table sizes, but the graphs were almost indistinguishable from the ones shown. From the figure we see the significant improvement of moving form one to more tables. For two tables nearly all of the $r/2$ heaviest keys are stored. For three tables this number increases to about $.88r$. Of the r heaviest keys, about 84% are stored using two tables and 95% using three tables.

The only difference noted between the graphs for different table sizes is that the exact point were the curves start to fall from 1 changes, converging towards a fixed value as r grows. For example, for the "two tables" scheme the curve starts to fall off at a distance before 0.5 that seems proportional to $\sqrt{1/r}$.

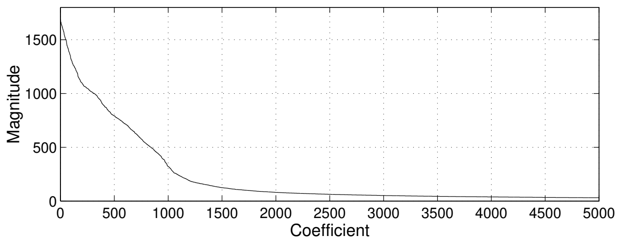

Fig. 2: Largest 5000 magnitudes of 67615 wavelet coefficients of the Lena image.

3.1 Application

To examine the practicality of our dictionaries we turn to the real world example of lossy image compression using wavelets. Today most state of the art image coders are based on wavelets, e.g., JPEG2000. The wavelet transform has the ability to efficiently approximate non-linear and non-stationary signals with coefficients whose magnitudes, in sorted order, decay rapidly towards zero. This is illustrated in Fig. 2. The figure shows the sorted magnitudes of the wavelet coefficients for the Lena image, a standard benchmark in image processing, computed using Daubechies second order wavelets. Thresholding the wavelet coefficients by a small threshold, i.e., setting small valued coefficients to zero, introduces only a small *mean squared error* (MSE) while leading to a sparse representation that can can be exploited for compression purposes. The main idea of most wavelet based compression schemes is to keep the value and position of the r coefficients of largest magnitude. To this end many advanced schemes, such as zero three coding, have been developed. All these schemes have in common that they do not support access to a single pixel without decoding significant portions of the image. Recently, interest in fast random access to decoded data, accessing only a few wavelet coefficients, has arisen [9, 10, 15]. We believe that our lossy dictionaries are well suited for this purpose.

We use the coefficients of Fig. 2 as an example of real world data, and store them in a two table lossy dictionary of total table size $r = 2^{11}$ and using a simple family of hash functions. Specifically, we use hash functions of the form

$$h(k) = ((a_2k^2 + a_1a_2k + a_0) \bmod p) \bmod r/2,$$

where p is a prime larger than any key, $0 < a_0, a_1, a_2 < p$ and a_1 is even. A corresponding quotient function is

$$q(k) = 2(((a_2k^2 + a_1a_2k + a_0) \bmod p) \operatorname{div} r/2) + k \bmod 2\ .$$

Again, 10^4 iterations were made, selecting random functions from the above family using C's `rand` function. The graph of $p_{r,\alpha r}$ is indistinguishable from that in Fig. 1. For our application, we obtain an MSE of 200, which is 27% more than the MSE when storing the r coefficients of highest magnitude. This difference would be difficult at best to detect in the reconstructed image. The previously mentioned family of [7] had somewhat worse performance. Using three tables reduces the MSE increase to a mere 1%.

4 Conclusion

We have introduced the concept of lossy dictionaries and presented a simple and efficient data structure implementing a lossy dictionary. Our data structure combines very efficient lookups and near-optimal space utilization, and thus seems a promising alternative to previously known data structures when a small percentage of false negatives is tolerable.

Though simple and efficient hash functions seem to work well in practice with our data structure, the challenge of finding such families that provably work well remains. Furthermore, the last two graphs in Fig. 1 are not completely understood, and the same is true for the insertion heuristic for three or more tables.

Acknowledgement We thank Theis Rauhe and Stephen Alstrup for helpful discussions on the construction of our two table data structure.

References

[1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[2] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640 (electronic), 1999.

[3] Harry Buhrman, Peter Bro Miltersen, Jaikumar Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 449–458. ACM Press, New York, 2000.

[4] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. System Sci.*, 18(2):143–154, 1979.

[5] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC '78)*, pages 59–65. ACM Press, New York, 1978.

[6] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial optimization.* John Wiley & Sons Inc., New York, 1998. A Wiley-Interscience Publication.

[7] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997. doi:10.1006/jagm.1997.0873.

[8] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. Assoc. Comput. Mach.*, 31(3):538–544, 1984.

[9] Insung Ihm and Sanghun Park. Wavelet-based 3D compression scheme for very large volume da ta. *Graphics Interface*, pages 107–116, June 1998.

[10] Tae-Young Kim and Yeong Gil Shin. An efficient wavelet-based compression method for volume rendering. In *Seventh Pacific Conference on Computer Graphics and Applications*, pages 147–156, October 1999.

[11] George Marsaglia. The Marsaglia random number CDROM including the diehard battery of tests of randomness. http://stat.fsu.edu/pub/diehard/.

[12] Rasmus Pagh. Low Redundancy in Static Dictionaries with $O(1)$ Lookup Time. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP '99)*, volume 1644 of *Lecture Notes in Computer Science*, pages 595–604. Springer-Verlag, Berlin, 1999.

[13] Rasmus Pagh. On the Cell Probe Complexity of Membership and Perfect Hashing. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pages ???–??? ACM Press, New York, 2001.

[14] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. Manuscript, 2001.

[15] Flemming Friche Rodler. Wavelet based 3D compression with fast random access for very large volume data. In *Seventh Pacific Conference on Computer Graphics and Applications*, pages 108–117, Seoul, Korea, October 1999.

[16] Robert Sedgewick. *Algorithms*. Addison-Wesley Series in Computer Sience. Addison-Wesley Publishing Company, Inc., second edition edition, 1988.

[17] Alan Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS '89)*, pages 20–25. IEEE Comput. Soc. Press, Los Alamitos, CA, 1989.