Succincter

Mihai Pătraşcu*
MIT

Abstract

We can represent an array of n values from $\{0,1,2\}$ using $\lceil n \log_2 3 \rceil$ bits (arithmetic coding), but then we cannot retrieve a single element efficiently. Instead, we can encode every block of t elements using $\lceil t \log_2 3 \rceil$ bits, and bound the retrieval time by t. This gives a linear trade-off between the redundancy of the representation and the query time.

In fact, this type of linear trade-off is ubiquitous in known succinct data structures, and in data compression. The folk wisdom is that if we want to waste one bit per block, the encoding is so constrained that it cannot help the query in any way. Thus, the only thing a query can do is to read the entire block and unpack it.

We break this limitation and show how to use recursion to improve redundancy. It turns out that if a block is encoded with two (!) bits of redundancy, we can decode a single element, and answer many other interesting queries, in time logarithmic in the block size.

Our technique allows us to revisit classic problems in succinct data structures, and give surprising new upper bounds. We also construct a locally-decodable version of arithmetic coding.

1 Introduction

1.1 Motivation

Can we represent data close to the information-theoretic minimum space, and still answer interesting queries efficiently? Two basic examples can showcase the antagonistic nature of compression and fast queries:

1. Suppose we want to store a bit-vector $A[1\,..\,n]$, and answer partial sums queries: $\text{RANK}(k)$, which asks for $\sum_{i=1}^{k} A[i]$; and $\text{SELECT}(k)$, which asks for the index of the k-th one in the array.

One the one hand, we must store some summaries (e.g. partial sums at various points) to support fast queries. On the other hand, a summary is, almost by definition, redundant. If we store a summary for every block of t bits, it would appear that the query needs to spend time proportional to t, because no guiding information is available inside the block.

2. Suppose we want to represent an array $A[1\,..\,n]$ of "trits" ($A[i] \in \{0,1,2\}$), supporting fast access to single elements $A[i]$. We can encode the entire array as a number in $\{0,\dots,3^n-1\}$, but the information about every trit is "smeared" around, and we cannot decode one trit without decoding the whole array.

Generalizing trits to symbols drawn independently from a distribution with entropy H, we can use arithmetic coding to achieve $n \cdot H + 1$ bits on average, but information about each element is spread around in the encoding. At the other extreme, we can use Huffman coding to achieve $n \cdot \big(H + O(1)\big)$ bits[1] of storage, which represents every element in "its own" memory bits, using a prefix-free code.

The natural solutions to these problems gravitate towards a linear trade-off between redundancy and query time. We can store a summary for every block of t elements (in problem 1.), or we can "round up" the entropy of every block of t symbols to an integral number of bits (in problem 2.). In both cases, we are introducing redundancy of roughly $\frac{n}{t}$, and the query time will be proportional to t.

It is not hard to convince oneself that a linear trade-off is the best possible. If we store t elements with at most $O(1)$ bits of redundancy, we need a super-efficient encoding that is essentially fixed due the entropy constraint. Because the encoding is so constrained, it would appear that it cannot be useful beyond representing the data itself. Then, the only way to work with such a super-efficient encoding is to decode it entirely, forcing query time proportional to t.

In this paper, we show that this intuition is false: we can use recursion even inside a super-efficient encoding (if we are allowed *two* bits of redundancy). Instead of decoding t elements to get to one trit, local decoding can be supported in logarithmic time. This extends to storing an entire augmented tree succinctly, so we can solve $\text{RANK}/\text{SELECT}$ in

*Supported by a Google Research Award and by MADALGO - Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

[1]More precisely, Gallager [11] showed that the redundancy is at most $p_{\max} + 0.086$ bits per element, where $p_{\max}$ is the maximum probability of an element.

logarithmic time. At every node of the tree, we can achieve something nontrivial (store the sum of its subtree), while introducing just $\frac{1}{t}$ bits of redundancy.

1.2 Succinct(er) Data Structures

In the field of succinct data structures, the goal is to construct data structures that use space equal to the information theoretic minimum plus some redundancy $\mathcal{R}$, while supporting various types of queries. The field has been expanding at a remarkable rate in the past decade, exploring a wide variety of problems and queries.

All of these structures, however, exhibit a linear trade-off between redundancy and query time. Typically the results are stated for constant query time, and achieve a fixed redundancy close to linear, most often around $O(\frac{n}{\lg n})$. At a high enough level of abstraction, this comes from storing $\varepsilon \lg n$ elements by an optimal encoding, and using a precomputed lookup table of size $O(n^{\varepsilon})$ to decode them in constant time. It can be seen that for any constant query time, the redundancy will not improve asymptotically (we can only look at a constant number of blocks).

For many natural problems in the field, our technique can achieve redundancy $O(n/\text{poly}\log n)$ with constant running times, for any desired $\text{poly}\log$. This suggests we have to rethink our expectations and approaches when designing succinct data structures.

Surprisingly, our constructions are often easier than the fixed-redundancy data structures they are replacing. It is not uncommon for succinct data structures to group elements into chunks, group chucks in superchunks, and finally group superchunks in megachunks. Each level has different lookup tables, and different details in the implementation. By having to do recursion for an unbounded number of levels, we are forced to discover a clean and uniform way to do it.

The following are a few results that follow from our technique. We believe however that the main value of this paper is to demonstrate that redundancy can be improved through recursion. These particular results are illustrations.

Locally decodable arithmetic codes. Our toy problem of storing ternary values can be generalized to representing an array of n elements with zeroth-order entropy H. Matching zeroth-order entropy is needed at the bottom of almost any technique attempting to match higher order entropy (including, for example, LZ77, the Burrows-Wheeler transform, JPEG, and MP3). Arithmetic coding can have a notable advantage over Huffman coding at low rates. For example, Shannon estimated the entropy of English to be 1.3 bits/letter, a rate at which a constant redundancy per letter can increase the encoding size by a significant percentage.

Mitzenmacher [17] uses arithmetic coding to compress Bloom filters, an application that critically requires local decodability. He asks for "a compression scheme that also provided random access," noting that "achieving random access, efficiency, and good compression simultaneously is generally difficult."

Our results give a version of locally-decodable arithmetic codes, in which the trade-off between local access time and redundancy is exponential:

Theorem 1. *Consider an array of n elements from an alphabet Σ, and let $f_\sigma > 0$ be the number of occurrences of letter σ in the array. On a RAM with cells of $\Omega(\lg n)$ bits, we can represent the array with:*

$$O(|\Sigma| \lg n) \;+\; \sum_{\sigma \in \Sigma} f_\sigma \log_2 \frac{n}{f_\sigma} \;+\; n \Big/ \Big(\frac{\lg n}{t}\Big)^t \;+\; \tilde{O}(n^{3/4})$$

bits of memory, supporting single-element access in $O(t)$ time.

Observe that $\sum f_\sigma \log_2 \frac{n}{f_\sigma}$ is the empirical entropy of the array. Thus, if the elements are generated by a memoryless process with entropy H, the expected space is $n \cdot H$, plus redundancy decreasing exponentially in t, plus $O(|\Sigma|)$ words needed to represent the distribution.

Bit-vectors with RANK/ SELECT**.** The problem of supporting RANK and SELECT on bit vectors is the bread-and-butter of succinct data structures, finding use in most other data structures (for representing trees, graphs, suffix trees / suffix arrays etc). Thus, the redundancy needed for this problem has come under quite a bit of attention.

The seminal papers of Jacobson [15] from FOCS'89, and Clark and Munro [5] from SODA'96 gave the first data structures using space $n + o(n)$ and constant query time. These results were later improved [19, 21, 26].

In several applications, the set of ones is not dense in the array. Thus, the problem was generalized to storing an array $A[1\,..\,u]$, containing n ones and $u - n$ zeros. The optimal space is $B = \lg \binom{u}{n}$. Note that the problem can solve predecessor search, by running $\text{SELECT}(\text{RANK}(i))$. From the lower bounds of [25], it follows that constant running times are only possible when the universe is not too much larger than n, in particular, $u = n \cdot \text{poly}\log n$. Thus, succinct data structures have focused on this regime of parameters.

Pagh [23] achieved space $B + O(n \cdot \frac{(\lg\lg n)^2}{\lg n})$ for this sparse problem. Recently, Golynski et al. [13] achieved $B + O(n \cdot \frac{\lg\lg u}{\lg^2 n})$. Finally, Golynski et al. [14] have achieved space $B + O(n \cdot \frac{\lg\lg n \cdot \lg(u/n)}{\lg^2 n})$. That paper conjectures that their redundancy is the best possible in constant time.

Here, we disprove their conjecture, and show that any redundancy $O(n/\text{poly}\log n)$ is achievable.

Theorem 2. *On a RAM with cells of $\Omega(\lg u)$ bits, we can represent an array $A[1\,..\,u]$ with n ones and $u-n$ zeros using $\log_2\binom{u}{n} + \frac{u}{(\lg u\ /t)^t} + \tilde{O}(u^{3/4})$ bits of memory, supporting* RANK *and* SELECT *queries in $O(t)$ time.*

The RANK/SELECT problem has also seen a lot of work in lower bounds [10, 16, 14, 12], particularly bounds applying to "systematic encodings." In this model, the bit vector is stored separately in plain form, and the succinct data structure consists of a sublinear size index on the side. Under this requirement, the best achievable redundancy with query time t is $\frac{n}{t\cdot\mathrm{poly}\lg n}$, i.e. the linear trade-off between redundancy and query time is the best possible. Our results demonstrate the significant power of not storing data in plain form.

Dictionaries. The dictionary problem is to store a set S of n elements from a universe u, and answer membership queries (is $x \in S$?) efficiently. Improving space has long been a primary goal in the study of dictionaries. For example, classic works put a lot of effort into analyzing linear probing when the table is close to capacity (if a $1-\varepsilon$ fraction of the cells are used, how does the "constant" running time degrade with ε?). Another trend is to study weaker versions of dictionaries, in the hope of saving space. Examples include the well known Bloom filters [1, 3, 22], and dictionaries that support retrieval only, sometimes called "Bloomier filters" [6, 4, 18].

The famous FKS dictionaries [8] were the first to solve the dictionary problem in linear space, in the sense of using $O(n)$ cells of size $\lg u$, while supporting queries in $O(1)$ time in the worst-case. Many data structures with similar performance have been suggested; in particular, cuckoo hashing [24] uses $(2+\varepsilon)n$ memory words.

Brodnik and Munro [2] were the first to approach the entropy bound, $B = \lg\binom{u}{n}$. Their data structure used space $B + O(B/\lg\lg\lg u)$, and they suggested that getting significantly more sublinear bounds might not be possible without "a more powerful machine model."

Pagh [23] gave the best known bound, achieving $B + O(n\frac{(\lg\lg n)^2}{\lg n})$. In fact, his algorithm is reduction to the RANK problem in vectors of size $u = n\cdot\mathrm{poly}\log n$. Thus, our results immediately imply dictionaries with redundancy $B + O(n/\lg^c n)$, for any constant c.

Balanced parentheses. Our techniques imply results identical to RANK/SELECT for the problem of storing a string of $2n$ balanced parentheses, and answering two queries:

MATCH(k) : find the parenthesis that matches the one on position k.

ENCLOSING(k) : find the open parenthesis that encloses most tightly the one on position k.

The optimal space is given by the Catalan number: $\lg\left(\frac{1}{n+1}\binom{2n}{n}\right) = 2n - O(\lg n)$. Due to the natural association between tree structures and balanced parentheses, this problem has been the crucial building block in most succinct tree representations.

A generic transformation. In fact, all of our results are shown via a generic transformation, converting a broad class of data structures based on augmented search trees into succinct data structures. It seems likely that such a broad result will have applications beyond the ones explored in this paper.

The reader is referred to §4 for formal details about the class of augmented search trees handled by our transformation.

1.3 Technical Discussion

Our goal is to improve redundancy through recursion, as opposed to a linear scan. A naïve view for how recursion might work for the trits problem is the following. We take w trits, which have entropy $w\log_2 3$, and "extract" some M bits of information (e.g. $M = \lfloor w\log_2 3\rfloor$), which we store. Then, the remaining $\delta = w\log_2 3 - M$ bits of information are passed to the second level of the recursion. At the second level, we aggregate w blocks, for which we must store $w\delta$ bits of information. We store some M' bits (e.g. $M' = \lfloor w\delta\rfloor$), and pass $\delta' = w\delta - M'$ bits to the second level, etc.

Since we are not willing to waste a bit of redundancy per block, δ may not be an integer. Unfortunately, "passing some fractional bits of information" is an intuition that does not render itself to any obvious implementation.

Our solution for passing a fractional number of entropy bits is elegant and, in hindsight, quite natural. We will approximate δ by $\log_2 K$, where K is an integer. This means that passing δ bits of information is almost the same as passing a number in $\{0,\ldots,K-1\}$. This approach introduces redundancy ("unused entropy") of $\log_2 K - \delta$ bits, a quantity that depends on how close δ is to a logarithm of an integer. Note however, that if K is the best approximation, δ is sandwiched between $\log_2(K-1)$ and $\log_2 K$. Thus, if we choose δ large enough, there is always some K that gives a good approximation.

At the second level of recursion, the problem will be to represent an array of n/w values from $\{0,\ldots,K-1\}$. This is just a generalization of the ternary problem to an arbitrary universe, so the same ideas apply recursively.

It should be noted that the basic technique of storing a certain number of bits and passing the "spill" to be stored separately, is not new. It was originally introduced by Munro et al. [20], and is the basis of the logarithmic improvement in redundancy of Golynski et al. [13] (see the "informative encodings" of that paper). Unfortunately, the techniques in these papers do no allow recursive composition, which is the key to our results.

In §2, we formally define the concept of *spill-over representations*, and uses it to prove Theorem 4 for representing an array of trits.

For less trivial applications, we need to compose variable-length representations without losing entropy. For example, we may want to store the sum of n numbers, and then the numbers themselves, by an efficient encoding that doesn't store the sum redundantly. With the right view, we can give a very usable lemma performing this composition; see §3. Finally, §4 uses this lemma to derive our main results in succinct data structures.

2 Spill-Over Representations

Assume we want to represent a value from a set $\mathcal{X}$. Any representation in a sequence of memory bits will use at least $\lceil \log_2 |\mathcal{X}| \rceil$ bits in the worst case, so it will have a redundancy of almost one bit if $|\mathcal{X}|$ is not close to a power of two. To achieve a redundancy much smaller than one for *any* value of $|\mathcal{X}|$, we must first define a model of "representation" where this is possible.

A *spill-over* representation consists of M memory bits (stored in our random-access memory), plus a number in $\{0, \dots, K-1\}$ that is stored by some outside entity. This number is called the *spill*, and K is called the *spill universe*. Observe that the spill-over representation is capable of representing $K \cdot 2^M$ distinct values. When using such a representation to store an element in $\mathcal{X}$, with $|\mathcal{X}| < K \cdot 2^M$, we define the redundancy of the representation to be $\log_2(K \cdot 2^M) - \log_2 |\mathcal{X}| = M + \log_2 K - \log_2 |\mathcal{X}|$. We can think of this as entropy wasted by the representation.

When talking about algorithms that access a spill-over representation, we assume the spill is provided by the outside entity for free, and the algorithm may access the memory bits by random access in the word memory.

As the most fundamental example of a spill-over representation, we have the following:

Lemma 3. *Consider an arbitrary set $\mathcal{X}$, and fix $r \le |\mathcal{X}|$. We can represent an element of $\mathcal{X}$ by a spill-over encoding with a spill universe K satisfying $r \le K \le 2r$, and redundancy at most $\frac{2}{r}$ bits.*

Proof. We must choose M and K carefully to achieve our redundancy bound. Specifically, we choose M to satisfy $2^M \cdot r \le |\mathcal{X}| \le 2^{M+1} \cdot r$. This fixes the spill universe to be $K = \lceil \frac{|\mathcal{X}|}{2^M} \rceil$; observe that $r < K \le 2r$.

Any injective map of $\mathcal{X}$ into $\{0,1\}^M \times \{0, \dots, K-1\}$ defines a spill-over scheme. For example, we can divide an index in $\mathcal{X}$ by 2^M, store the remainder as M memory bits, and pass the quotient as the spill. This is decodable by $O(1)$ arithmetic operations.

Our spill-over scheme is capable of representing $2^M \cdot K$ different values, as opposed to the $|\mathcal{X}|$ values required. Thus, we have a redundancy of:

$$\begin{aligned}
\log_2\left(\frac{K \cdot 2^M}{|\mathcal{X}|}\right) &= \log_2\left(\frac{\lceil \frac{|\mathcal{X}|}{2^M} \rceil \cdot 2^M}{|\mathcal{X}|}\right) \\
&\le \log_2\left(\frac{(\frac{|\mathcal{X}|}{2^M} + 1) \cdot 2^M}{|\mathcal{X}|}\right) \\
&= \log_2\left(1 + \frac{2^M}{|\mathcal{X}|}\right) \le \log_2\left(1 + \frac{1}{r}\right) \le \frac{2}{r} \quad \square
\end{aligned}$$

2.1 Application: Storing Trits

We now show how to compose spill-over representations from Lemma 3 recursively, yielding the following bounds for our toy problem of storing n ternary values:

Theorem 4. *On a RAM with w-bit cells, we can represent an array $A[1\,..\,n]$ with $A[i] \in \{0,1,2\}$ using $\lceil n \log_2 3 \rceil + \frac{n}{(w/t)^t} + \operatorname{poly}\log n$ bits of memory, while supporting single-element accesses in $O(t)$ time.*

Proof. We first group elements into blocks of w. A block takes values in a set of size 3^w. We apply Lemma 3 with some parameter r to be determined, and store each block as M_0 memory bits and a spill in a universe $K_0 \in [r, 2r]$. Given the spill, we can read the $O(w)$ memory bits and decode in constant time:

- first assemble the spill and the memory bits, multiplying the spill by 2^{M_0}, and adding them together.
- now extract the ith trit, dividing by 3^i, and taking the number modulo 3.

In practice, we want to avoid division by precomputing a table with the multiplicative inverses of 3^i. Note that we do not need explicit pointers to each block, since the offset at which the memory bits of a block are stored is equal to M_0 times the block index.

Now let $B = \Theta(\frac{w}{\lg r})$, with $B \ge 2$. At the second level of recursion, we store the spills of B consecutive blocks. There are $K_0^B \le (2r)^B = 2^{O(w)}$ choices for the bottom-level spills. Using Lemma 3, we can represent this data as $M_1 \le \log_2(K_0^B) = O(w)$ memory bits, and a spill in universe $K_1 \in [r, 2r]$. Note that the assumption $r \le |\mathcal{X}|$ made by the lemma is indeed satisfied, because $|\mathcal{X}| = K_0^B \ge r^2$. Since Lemma 3 is applied in identical conditions, K_1 and M_1 will be identical for all level-1 blocks.

We can continue to apply this scheme recursively, storing B spills in universe K_1, generating a spill in universe $K_2 \in [r, 2r]$, etc. We do this for t levels, supporting access queries in $O(t)$ word probes. At the end of the recursion, we store each of the final spills with $\lceil \log_2 K_t \rceil$ bits, paying one bit of redundancy per spill. Note that the size of the final scheme is a telescoping sum of the sizes at intermediate levels of recursion, i.e. the final redundancy is simply the sum of the redundancies introduced at each step. This gives:

$$\begin{aligned}\mathcal{R} &= O\left(\frac{n/w}{r} + \frac{n/(Bw)}{r} + \cdots + \frac{n/(B^t w)}{r} + \frac{n}{B^t w}\right) \\ &= O\left(\frac{n}{wr} + \frac{n}{w \cdot B^t}\right)\end{aligned}$$

To balance the two terms (the redundancy of the spill-over representations, versus the final redundancy of one bits per spill), let $r = B^t$. By our choice of B, we have $B = O(\frac{w}{\lg r}) = O(\frac{w}{t \lg B}) = O(\frac{w/t}{\lg(w/t)})$. We thus have $r = B^t = \left(\frac{w}{t}\right)^{\Omega(t)}$. Adjusting constants, we have a redundancy of $\mathcal{R} \le \frac{n}{(w/t)^t}$, and query time $O(t)$.

As a header of the data structure, we need to store the values K_i and M_i for every level of the recursion, which are needed to navigate the data structure. This adds $O(\lg^2 n)$ bits to the redundancy. $\square$

3 Composing Variable-Length Encodings

As we have just seen, spill-over encodings of fixed length can be composed easily in a recursive fashion. However, in less trivial applications, we need to compose variable-length encodings without losing entropy. For an illustration of the concept, assume we want to store an array $A[1\,..\,n]$ of bits, such that we can query both any element $A[i]$, and the sum of the entire array: $X = \sum_{j=1}^{n} A[j]$. If we choose the trivial encoding as n bits, querying the sum will take linear time.

Conceptually, we must first store the sum, followed by the array itself, represented by an *efficient* encoding that uses knowledge of the sum (i.e. does not contain any redundant information about the sum). This should not lose entropy, since $H(X) + H(A|X) = n$.

The trouble is that the bound $H(X)$ can only be achieved in expectation, by some kind of arithmetic code. Furthermore, since $H(A|X)$ is a function of X, the parameters of a spill-over representation must also vary with X. Thus, we need to piece together some kind of arithmetic code for X, with a variable-length spill-over representation of A whose size depends on X. In the end, however, we should obtain a fixed-length encoding, since the overall entropy is n bits.

The following lemma formalizes this intuition, in a statement that has been crafted carefully for ease of use in applications:

Lemma 5. *Assume we have to represent a variable $x \in \mathcal{X}$, and a pair $(y_M, y_K) \in \{0,1\}^{M(x)} \times \{0, \dots, K(x) - 1\}$. Let $p(x)$ be a probability density function on $\mathcal{X}$, and $K(\cdot), M(\cdot)$ be non-negative functions on $\mathcal{X}$ satisfying:*

$$(\forall)x \in \mathcal{X}: \quad \log_2 \frac{1}{p(x)} + M(x) + \log_2 K(x) \le H \quad (1)$$

We can design a spill-over representation for x, y_M and y_K with the following parameters:

- *the spill universe is $K_\star$ with $K_\star \le 2r$, and the memory usage is $M_\star$ bits.*
- *the redundancy is at most $\frac{4}{r}$ bits, i.e. $M_\star + \log_2 K_\star \le H + \frac{4}{r}$.*
- *if the word size is $w = \Omega\big(\lg|\mathcal{X}| + \lg r + \lg \max K(x)\big)$, x and y_K can be decoded with $O(1)$ word probes. The input bits y_M can be read directly from memory, but only after y_K is retrieved.*
- *given a precomputed table of $O(|\mathcal{X}|)$ words that only depends on the input functions K, M and p, decoding x and y_K takes constant time on the word RAM.*

In §3.1, we describe the main details of the construction. To make the construction implementable, we need to tweak the distribution $p(\cdot)$ to avoid pathologically rare events; this is detailed in §3.2. Finally, in §3.3 we describe the constant-time decoding procedure, which relies on a cute algorithmic trick. Note that a table of size $O(|\mathcal{X}|)$ is optimal, since the lemma is instantiated for three arbitrary functions on $\mathcal{X}$.

3.1 The Basic Construction

The design of our data structure is remarkably simple. First, let $M_{\min} = \min_{x \in \mathcal{X}} M(x)$. We can decrease each $M(x)$ to $M_{\min}$, by encoding $M(x) - M_{\min}$ bits of memory into the spill. This has the technical effect that we may not access y_M prior to decoding y_K, as stated in the lemma. From now on, we assume y_M always has $M_{\min}$ bits, and y_K comes from a universe of $K'(x) = K(x) \cdot 2^{M(x) - M_{\min}}$.

Now let Z be the set of possible pairs (x, y_K).

Claim 6. *We have $\log_2 |Z| + M_{\min} \le H$.*

Proof. We will show $|Z| \le \max_{x \in \mathcal{X}} \frac{K'(x)}{p(x)}$. Since $\log_2 \frac{1}{p(x)} + \log_2 K'(x) + M_{\min} \le H$ for all x, it follows that $\log_2 |Z| + M_{\min} \le H$.

Suppose for contradiction that $|Z| > \frac{K'(x)}{p(x)}$ for all $x \in \mathcal{X}$. Then, $K'(x) < |Z| \cdot p(x)$, for all x. We have $|Z| = \sum_{x \in \mathcal{X}} K'(x) < \sum_{x \in \mathcal{X}} \big(|Z| \cdot p(x)\big) = |Z|$, which gives a contradiction. $\square$

Though the proof of this lemma looks like a technicality, the intuition behind it is quite strong. The space of encodings Z is partitioned into equivalence classes $(x, \star)$, giving each element x a fraction equal to $K(x)/|Z|$. But $\log_2 \frac{1}{p(x)} + \log_2 K(x) \approx H - M_{\min}$, so $K(x)/p(x)$ is roughly *fixed*. Thus, the fraction of space assigned to x is roughly $p(x)$, which is exactly how arithmetic coding operates (partitioning the real interval $[0, 1]$).

To complete the representation, we apply Lemma 3 to the set Z. The resulting spill is passed along as the spill of our data structure, and the memory bits are stored at the beginning of the data structure, followed by the $M_{\min}$ bits of y_M. The only redundancy introduced is by this application of Lemma 3, i.e. $\frac{2}{r}$ bits.

For this construction to be efficiently decodable, we must at the very least be able to manipulate a value of Z in constant time, i.e. have a word size of $w = \Omega(\lg |Z|)$. To understand this requirement, we bound $\log_2 |Z| \le H - \min M(x)$. For every x, we are free to increase $M(x)$, padding with zero bits, up to the maximum integer satisfying $M(x) \le H - \log_2 K(x) - \log_2 \frac{1}{p(x)}$. Thus, $M(x) > H - \log_2 \frac{K(x)}{p(x)} - 1$. This implies the bound $\log_2 |Z| \le \log_2 \frac{\max K(x)}{\min p(x)} + 1$, i.e. $|Z| = O\big(\frac{\max K(x)}{\min p(x)}\big)$. The statement of the lemma already assumes $w = \Omega(\lg \max K(x))$, since $K(x)$ is the universe of the input spill. Thus, it remains to ensure that $w = \Omega(\lg \frac{1}{\min p(x)})$.

3.2 Handling Rare Events

Unfortunately, some values $x \in \mathcal{X}$ can be arbitrarily rare, and in fact, $\min p(x)$ is prohibitively small even for natural applications. Remember our example of representing an array $A[1\,..\,n]$ of bits, together with its sum $x = \sum_{j=1}^{n} A[i]$. We have $\min p(x) = p(n) = 2^{-n}$, which means our algorithm wants to manipulate spills of $\Omega(n)$ bits. A word size of $\Theta(n)$ bits is an unrealistic assumption.

Our strategy will be to tweak the distribution $p(\cdot)$ slightly, increasing $\min p(x)$ towards $\frac{1}{|\mathcal{X}|}$, while not losing too much entropy if we code according to this false distribution. (In information-theoretic terms, the Kullback-Leibler divergence between the distributions must be small.)

Formally, we tweak the distribution by adding $\frac{1}{|X|\cdot r}$ to every probability, and then normalizing:

$$p'(x) = \Big(p(x) + \frac{1}{|X| \cdot r}\Big) \Big/ \Big(1 + \frac{1}{r}\Big)$$

Observe that $\sum_{x\in X} p'(x) = \big(\sum_{x\in X} p(x) + |X| \cdot \frac{1}{|X|\cdot r}\big) / \big(1+\frac{1}{r}\big) = 1$, so $p'(\cdot)$ indeed defines a distribution.

We now have $\min p'(x) \ge \frac{1}{|\mathcal{X}|\cdot r} / (1 + \frac{1}{r}) \ge \frac{1}{2r\cdot|\mathcal{X}|}$. This requires the word size to be $w = \Omega(\lg r + \lg |\mathcal{X}|)$, a reasonable assumption made by the lemma. (In our example with an n-bit array, $|\mathcal{X}| = n+1$, so we require $w = \Omega(\lg r + \lg n)$, instead of $\Omega(n)$ are before.)

Finally, we must show that the entropy bound in (1) is not hurt too much if we code according to $p'(\cdot)$ instead of $p(\cdot)$. Note that:

$$\begin{aligned} \log_2 \tfrac{1}{p'(x)} \;&\le\; \log_2 \tfrac{1}{p(x)/(1+1/r)} \\ &=\; \log_2 \tfrac{1}{p(x)} + \log_2\big(1+\tfrac{1}{r}\big) \;\le\; \log_2 \tfrac{1}{p(x)} + \tfrac{2}{r} \end{aligned}$$

We can replace (1) with the following weaker guarantee:

$$\begin{aligned} &M(x) + \log_2 K(x) + \log_2 \tfrac{1}{p'(x)} \\ &\quad\le\; M(x) + \log_2 K(x) + \log_2 \tfrac{1}{p(x)} + \tfrac{2}{r} \;\le\; H + \tfrac{2}{r} \end{aligned}$$

Combining with the construction from §3.1, which introduced another $\frac{2}{r}$ bits of redundancy, we have lost $\frac{4}{r}$ bits of redundancy overall.

3.3 Algorithmic Decoding

The heart of the decoding problem lies in recovering x and y_K based on an index in Z (output by Lemma 3). We could do this with a lookup in a precomputed table of size $|Z|$. Remember that we bounded $|Z| = O(\frac{\max K(x)}{\min p(x)}) = O(|\mathcal{X}| \cdot r \cdot \max K(x))$, which is not a strong enough bound for some applications. We now show how to use a table of size just $O(|\mathcal{X}|)$.

From the point of view of x, the space Z is partitioned into pieces of cardinality $K(x)$, and the query is to find the piece containing a given codeword. We are free to design the partition to make decoding efficient. First, we assign to each x a contiguous interval of Z. Let z_x be the left boundary of the interval assigned to x. Decoding x is equivalent to a predecessor search, locating the codeword among the values $\{z_1, \dots, z_{|\mathcal{X}|}\}$. Decoding y_K simply subtracts z_x from the codeword.

Unfortunately, the optimal bounds for predecessor search [25] are superconstant in the worst case. To achieve constant time, we must leverage our ability to choose the encoding: we must arrange the intervals in an order that makes predecessor search easy! While this sounds mysterious, it turns out that sorting the intervals by increasing length suffices.

Claim 7. *If intervals are sorted by length, i.e. $z_{i+1} - z_i \ge z_i - z_{i-1}$, predecessor search among the z_i's can be supported in constant time by a data structure of $O(|\mathcal{X}|)$ words.*

Proof. Let $f(\tau)$ be the smallest interval of length τ, i.e. $f(\tau) = \min\{i \mid z_{i+1} - z_i \ge \tau\}$. Consider the set of $z_{f(\tau)}$, for every τ a power of two. This set has $O(w)$ values, so we can store it in a fusion tree [9], and support predecessor search in constant time.

Say we have located the query between $z_{f(\tau)}$ and $z_{f(2\tau)}$. All intervals in this range have width between τ and 2τ, i.e. we have constant spread. In this case, predecessor search can be solved easily in constant time [7]: break the universe into buckets of size τ, which ensures at most one value per bucket, and at most three buckets to inspect until the predecessor is found. $\square$

4 Applications to Succinct Data Structures

4.1 Augmented Trees

As mentioned already, our results are based on a generic transformation of augmented B-trees to succinct data structures. For some $B \geq 2$, we define a class of data structures, *aB-trees*, as follows:

- The data structure represents an array $A[1\,..\,n]$ of elements from some alphabet Σ, where n is a power of B. The data structure is a B-ary tree with the elements of A in the leaves.
- Every node is augmented with a value from some alphabet Φ. The value of a leaf is a function of its array element, and the value of an internal node is a function $\mathcal{A}$ of the values of its B children, and the size of the subtree.
- The query algorithm examines the values of the root's children, decides which child to recurse to, examines all values of that node's children, recurses to one of them, etc. When a leaf is examined, the algorithm outputs the query answer. We assume the query algorithm spends constant time per node, if all values of the children are given packed in a word.

For instance, the RANK/SELECT problem has a standard solution through an aB-tree. The alphabet Σ is simply $\{0, 1\}$. Every internal node counts the sum of the leaves in its subtree (equivalently, the sum of its children), so $\Phi = \{0, \dots, n\}$. The queries can be solved in constant time per node if the values of all children are given packed in a word. This uses very standard ideas from word-level parallelism that we omit.

We aim to compress an aB-tree. A natural goal for the space an aB-tree should use is given by $\mathcal{N}(n, \varphi)$, defined as the number of instances of $A[1\,..\,n]$ such that the root is labeled with $\varphi \in \Phi$. Observe that we can write the following recursion for $\mathcal{N}(B \cdot n, \varphi)$:

$$\mathcal{N}(B \cdot n, \varphi) = \sum_{\varphi_1, \dots, \varphi_B : \mathcal{A}(\varphi_1, \dots, \varphi_B, n) = \varphi} \mathcal{N}(n, \varphi_1) \cdots \mathcal{N}(n, \varphi_B)$$

Indeed, any instance for the first half is valid, as long as its aggregate value combines properly with the aggregate of the second half.

To develop intuition for $\mathcal{N}$, observe that in the RANK/SELECT example, $\mathcal{N}(n, \varphi) = \binom{n}{\varphi}$, because φ was just the number of ones in the array. Our recursion becomes the following obvious identity:

$$\mathcal{N}(B \cdot n, \varphi) = \sum_{\varphi_1 + \cdots + \varphi_B = \varphi} \mathcal{N}(n, \varphi_1) \cdots \mathcal{N}(n, \varphi_B)$$

We will show the following general result:

Theorem 8. *Let $B = O(\frac{w}{\lg(n+|\Phi|)})$. We can store an aB-tree of size n with root value φ using $\log_2 \mathcal{N}(n, \varphi) + 2$ bits. The query time is $O(\log_B n)$, assuming precomputed look-up tables of $O(|\Sigma| + |\Phi|^{B+1} + B \cdot |\Phi|^B)$ words, which only depend on n, B and the aB-tree algorithm.*

Essentially, this result compresses the entire aB-tree with only two 2 bits of redundancy. The additional space of the look-up tables will not matter too much, since we construct many data structures that share them.

Application to RANK/SELECT. Say we want to solve RANK and SELECT in time $O(t)$, for an array of size U with N ones. As mentioned already, RANK and SELECT queries can be supported by an aB-tree, so Theorem 8 applies. If the aB-tree has size r, we have $|\Sigma| = 2$ and $|\Phi| = r + 1$.

Choose $B \geq 2$ such that $B \lg B = \frac{\varepsilon \lg U}{t}$, and let $r = B^t = \left(\frac{\lg U}{t}\right)^{\Theta(t)}$. We break the array into buckets of size r, rounding up the size of the last bucket. Each bucket is stored as a succinct aB-tree. Supporting RANK and SELECT inside such an aB-tree requires time $O(\log_B r) = O(t)$.

For each bucket, we store the the index in memory of the bucket's memory bits. Let $N_1, N_2, \dots$ be number of ones in each subarray. We store a partial sums vector for these values (to aid RANK), and a predecessor structure on the partial sums (to aid SELECT). We have at most U/r values from a universe of U, so the predecessor structure can support query time $O(t)$ using space $\frac{U}{r} \cdot r^{\Omega(1/t)} \leq \frac{U}{r} \cdot B = U/B^{t-1}$ words; see [25]. A query begins by examining these auxiliary structures, and then performing a query in the right aB-tree.

The components of the memory consumption are:

1. a pointer to each bucket, and the partial sums for the array $N_1, N_2, \dots$. These occupy $O(\frac{U}{r} \lg U) = O\left(\frac{U \lg U}{B^t}\right)$ bits.
2. the predecessor structure, occupying $O(U/B^{t-1})$ words. This dominates item 1., and is $U/B^{\Theta(t)}$ bits.
3. the succinct aB-trees, which occupy:

$$\begin{aligned} \textstyle\sum_i \left[\log_2 \binom{r}{N_i} + 2\right] \;&\leq\; \log_2 \textstyle\prod_i \binom{r}{N_i} + O\left(\frac{U}{r}\right) \\ &\leq\; \log_2 \binom{U+r-1}{N} + O\left(\frac{U}{r}\right) \;\leq\; \log_2 \binom{U}{N} + O\left(r + \frac{U}{r}\right) \end{aligned}$$

   bits. The redundancy can be rewritten as $r + \frac{U}{r} = O(\max\{\frac{U}{r}, \sqrt{U}\})$.
4. the look-up tables, of size $O((r+1)^{B+1} + (r+1)^B \cdot B) = 2^{O(tB \lg B)} = 2^{O(\varepsilon \lg U)} = U^{O(\varepsilon)}$. Setting ε a small enough constant, this contributes negligibly to the redundancy. The only limitation is $B \geq 2$, so we cannot reduce the lookup tables below $O(r^3)$ words. A redundancy of $\frac{U}{r} + O(r^3)$ can be written as $\max\{\frac{U}{r}, U^{3/4}\}$.

To summarize, we obtain a redundancy of $U/B^{\Theta(t)} + O(U^{3/4})$. Readjusting constants in t, the redundancy is $U/\left(\frac{\lg U}{t}\right)^t + O(U^{3/4})$.

Application to arithmetic coding. In this application, Σ is the alphabet that we wish to encode. Intuitively, a letter $\sigma \in \Sigma$ has $\log_2 \frac{n}{f_\sigma}$ bits of entropy. We round these values up to multiples of $\frac{1}{r}$, which only adds redundancy $\frac{n}{r}$ over all symbols.

We construct an aB-tree, in which internal nodes are augmented to store the entropy of the symbols in their subtree. If the aB-tree has size at most r, the total entropy is at most $O(r \lg n)$, so $|\Phi| = O(\lg r + \lg\lg n)$. The query algorithm is trivial: it just traverses the tree down to the leaf that it wants to retrieve, ignoring all nodes along the way.

By this definition, $\mathcal{N}(n, \varphi)$ is the number of n-letter sequences with total entropy exactly φ. But there are at most 2^φ such sequences, by an immediate packing argument. Thus, an aB-tree of size r having value φ at the root can be compressed to space $\varphi + 2$. We now proceed as in the previous example, breaking the array into n/r buckets of size r, and performing the same calculations for the space occupied by auxiliary structures.

4.2 Proof of Theorem 8

The proof is by induction on powers of B, aggregating B spill-over representations for aB-trees of size n/B into one for an aB-tree of size n. Let $K(n, \varphi)$ be the spill universe used for a data structure of size n and root label φ. Let $M(n, \varphi)$ be the memory bits used by such a representation.

Let r to be determined. We guarantee inductively that:

$$K(n, \varphi) \le 2r; \tag{2}$$

$$M(n, \varphi) + \log_2 K(n, \varphi) \le \log_2 \mathcal{N}(n, \varphi) + 4\frac{2n-1}{r}. \tag{3}$$

Consider the base case, $n = 1$. The alphabet Σ is partitioned into sets Σ_φ of array elements for which the leaf is labeled with φ. We have $\mathcal{N}(1, \varphi) = |\Sigma_\varphi|$. We use a spill-over encoding as in Lemma 3 to store an index into Σ_φ. The encoding will use a spill universe $K(1, \varphi) \le 2r$ and $M(1, \varphi)$ bits of memory, such that $M(1, \varphi) + \log_2 K(1, \varphi) \le \log_2 |\Sigma_\varphi| + \frac{2}{r}$. We store a look-up table that determines the array value based on the value φ and the index into Σ_φ. These look-up tables (for all φ) require space $O(|\Phi| + |\Sigma|)$.

For the induction step, we break the array into B subarrays of size n/B. Let $\widetilde{\varphi} = (\varphi_1, \dots, \varphi_B)$ denote the values at the root of each of the B subtrees. We recursively represent each subarray using $M(\frac{n}{B}, \varphi_i)$ bits of memory and spill universe $K(\frac{n}{B}, \varphi_i)$.

Then, all memory bits from the children are concatenated into a bit vector of size $M' = \sum_i M(\frac{n}{B}, \varphi_i)$, and the spills are combined into a superspill from the universe $K' = \prod_i K(\frac{n}{B}, \varphi_i)$. Since $\lg K' \le \lg\left((2r)^B\right) = O(B \lg r)$, we require that $B = O(w/\lg r)$, so that this superspill fits in a constant number of words. For every possible $\widetilde{\varphi}$, we precompute the partial sums of $M(\frac{n}{B}, \varphi_i)$, so that we know where the ith child begins in constant time. We also precompute the constant needed to extract the ith spill from the superspill. These tables require $O(B \cdot |\Phi|^B)$ words.

Summing the recursive guarantee (3) of every child, we have:

$$\log_2 K' + M' \;\le\; \log_2 \prod_i \mathcal{N}\Big(\frac{n}{B}, \varphi_i\Big) \;+\; 4 \cdot \frac{2n - B}{r}$$

Let $\varphi = \mathcal{A}(\varphi_1, \dots, \varphi_i, n)$ be the value at the root. Let $p(\cdot)$ be the distribution of $\widetilde{\varphi}$ given this value of φ, that is:

$$p(\widetilde{\varphi}) \;=\; \prod_i \mathcal{N}\Big(\frac{n}{B}, \varphi_i\Big) \Big/ \mathcal{N}(n, \varphi)$$

But then:

$$\log_2 K' + M' \;\le\; \log_2 \mathcal{N}(n, \varphi) - \log_2 \frac{1}{p(\widetilde{\varphi})} + 4 \cdot \frac{2n - B}{r}$$

This satisfies the entropy condition (1) of Lemma 5. We apply the lemma to represent $\widetilde{\varphi}$, the superspill, and the memory bits of the subarrays. We obtain a representation with spill universe $K_\star \le 2r$ and $M_\star$ memory bits, such that:

$$\begin{aligned} M_\star + \log_2 K_\star \;&\le\; \log_2 \mathcal{N}(n, \varphi) \;+\; 4\frac{2n - B}{r} \;+\; \frac{4}{r} \\ &\le\; \log_2 \mathcal{N}(n, \varphi) + 4 \cdot \frac{2n - 1}{r} \end{aligned}$$

The precomputed table required by Lemma 5 is linear in the support of $p(\cdot)$, which is $|\Phi|^B$. Such a table is stored for every distinct φ, giving space $|\Phi|^{B+1}$. We must have $B = O(\frac{w}{\lg |\Phi|})$.

This completes the induction step. To prove Theorem 8, we construct the above representation for the required size n, using the value $r = \frac{1}{8n}$. Note that this requires $B \le O\big(\frac{w}{\lg \max\{|\Phi|, r\}}\big)$.

At the root, the final spill is stored explicitly at the beginning of the data structure. Thus, the space is:

$$\lceil \log_2 K(n, \varphi) \rceil + M(n, \varphi) \;\le\; \log_2 \mathcal{N}(n, \varphi) + 2$$

The queries are easy to support. First, we read the final spill at the root. Then, we decode $\widetilde{\varphi}$ and the superspill from the representation of Lemma 5. The aB-tree query algorithm decides which child to follow recursively based on $\widetilde{\varphi}$. We extract the spill of that child from the superspill, and recurse. The constants needed to extract the spill and the position in memory of the child were stored in look-up tables.

5 Open Problems

It is an important open problem to establish whether our exponential trade-off between redundancy and query time is optimal. We conjecture that it is. Unfortunately, proving this seems beyond the scope of current techniques. The only lower bound for succinct data structures (without the systematic assumption) is via a rather simple idea of Gál and Miltersen [10], which requires that the data structure have an intrinsic error-correcting property. Such a property is not characteristic of our problems.

Even if the exponential trade-off cannot be improved, it would be interesting to establish where this trade-off "bottoms." Due to our need for large precomputed tables, the smallest redundancy that we can achieve is some $O(n^\alpha)$ bits, where α is a constant close to one (for instance, $\alpha = 3/4$ for RANK/SELECT). Can this redundancy be reduced to, say, $O(\sqrt{n})$, or hopefully even $O(n^\varepsilon)$?

Acknowledgments. This work was initiated while the author was visiting the Max Planck Institut für Informatik, Saarbrücken, in June 2005. I wish to thank Seth Pettie for telling me about the problem on that occasion, and for initial discussions. I also wish to thank Rasmus Pagh and Rajeev Raman for helping me understand previous work.

References

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

[2] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999. See also ESA'94.

[3] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proc. 10th ACM Symposium on Theory of Computing (STOC)*, pages 59–65, 1978.

[4] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The Bloomier filter: an efficient data structure for static support lookup tables. In *Proc. 15th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 30–39, 2004.

[5] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proc. 7th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, 1996.

[6] Erik D. Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Pătraşcu. De dictionariis dynamicis pauco spatio utentibus (lat. on dynamic dictionaries using little space). In *Proc. Latin American Theoretical Informatics (LATIN)*, pages 349–361, 2006.

[7] Erik D. Demaine, Thouis Jones, and Mihai Pătraşcu. Interpolation search for non-independent data. In *Proc. 15th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–523, 2004.

[8] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984. See also FOCS'82.

[9] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. See also STOC'90.

[10] Anna Gál and Peter Bro Miltersen. The cell probe complexity of succinct data structures. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 332–344, 2003.

[11] Robert G. Gallager. Variations on a theme by huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, 1978.

[12] Alexander Golynski. Optimal lower bounds for rank and select indexes. *Theoretical Computer Science*, 387(3):348–359, 2007. See also ICALP'06.

[13] Alexander Golynski, Roberto Grossi, Ankur Gupta, Rajeev Raman, and S. Srinivasa Rao. On the size of succinct indices. In *Proc. 15th European Symposium on Algorithms (ESA)*, pages 371–382, 2007.

[14] Alexander Golynski, Rajeev Raman, and S. Srinivasa Rao. On the redundancy of succinct data structures. In *Proc. 11th Scandinavian Workshop on Algorithm Theory (SWAT)*, 2008.

[15] Guy Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[16] Peter Bro Miltersen. Lower bounds on the size of selection and rank indexes. In *Proc. 16th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–12, 2005.

[17] Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking*, 10(5):604–612, 2002. See also PODC'01.

[18] Christian Worm Mortensen, Rasmus Pagh, and Mihai Pătraşcu. On dynamic range reporting in one dimension. In *Proc. 37th ACM Symposium on Theory of Computing (STOC)*, pages 104–111, 2005.

[19] J. Ian Munro. Tables. In *Proc. 16th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–40, 1996.

[20] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 345–356, 2003.

[21] J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001. See also FSTTCS'98.

[22] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal Bloom filter replacement. In *Proc. 16th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 823–829, 2005.

[23] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001. See also ICALP'99.

[24] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004. See also ESA'01.

[25] Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[26] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, 233–242, 2002.