Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 54 (2005) 3–23

Science of Computer Programming

www.elsevier.com/locate/scico

Mutable strings in Java: design, implementation and lightweight text-search algorithms

Paolo Boldi*, Sebastiano Vigna

Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Milano, Italy

Received 16 January 2004; received in revised form 7 May 2004; accepted 20 May 2004

Available online 17 July 2004

Abstract

The Java string classes, `String` and `StringBuffer`, lie at the extremes of a spectrum (immutable, reference based, and mutable, content based). Analogously, available text-search methods on string classes are implemented either as trivial, brute-force double loops, or as very sophisticated and resource-consuming regular-expression search methods. Motivated by our experience in data-intensive text applications, we propose a new string class, `MutableString`, which tries to get the right balance between extremes in both cases. Mutable strings can be in one of two states, *compact* and *loose*, in which they behave more like `String` and `StringBuffer`, respectively. Moreover, they support a wide range of sophisticated text-search algorithms with a very low resource usage and set-up time, using a new, very simple randomised data structure (a generalisation of Bloom filters) that stores an approximation from above of a lattice-valued function. Computing the function value requires a constant number of steps, and the error probability can be balanced with space usage. As a result, we obtain practical implementations of Boyer–Moore type algorithms that can be used with very large alphabets, such as Unicode collation elements. The techniques we develop are very general and amenable to a wide range of applications.

1. Introduction

The Java string classes, `String` and `StringBuffer`, lie at the extremes of a spectrum (immutable, reference based, and mutable, content based).

* Corresponding author.
E-mail addresses: boldi@acm.org (P. Boldi), vigna@acm.org (S. Vigna).
URLs: http://boldi.dsi.unimi.it/ (P. Boldi), http://vigna.dsi.unimi.it/ (S. Vigna).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.05.003

However, in several applications this dichotomy results in inefficient object handling, and in particular in the creation of many useless temporary objects. In very large data applications, with millions of alive objects, the cost of this inefficiency may be exceedingly high.

Another problem arises with a typical string usage pattern; in this common scenario, you start with a mutable string object, of an as-yet unknown length, and append, delete, insert and substitute characters; at a certain point in time, you end up with a string that will not be changed thereafter, and you would like to "freeze" its state.

To replicate this scenario using the standard Java string classes, you will most probably use a `StringBuffer` in the first phase, and then turn it into a `String`, by using the `toString` method of `StringBuffer`. Unfortunately, the first phase will be slowed down by the synchronisation of `StringBuffer` methods, whereas the majority of applications will not need synchronisation at all (or will accommodate their synchronisation needs at a higher level). Moreover, turning the `StringBuffer` into a `String` implies the creation of a new object.

Of course, one might simply decide not to turn the `StringBuffer` into a `String`, but this makes it impossible to use it in the same optimised way as an immutable string; even worse, it is impossible to use a `StringBuffer` in a collection, as it does not override the `equals()` method provided by `Object`.

This dissatisfaction with the behaviour of `String` and `StringBuffer` is well known in the Java community. For instance, the Altavista crawler, Mercator [12], has been written in Java, but the authors admit that one of their first steps was rewriting the standard Java string classes [13]. The authors have also experienced similar troubles when writing their web crawler, UbiCrawler [5], and later when indexing its results.

Sun itself is very aware of the problem: one of the suggestions given to people struggling to improve application performance reads as follows [15]:

> 11.1.3.14 Making immutable objects mutable like using StringBuffers
>
> Immutable objects serve a very good purpose but might not be good for garbage collection since any change to them would destroy the current object and create a new objects i.e., more garbage. Immutables by description, are objects which change very little overtime. But a basic object like String is immutable, and String is used everywhere. One way to bring down the number of objects created would be to use something like StringBuffers instead of Strings when String manipulation is needed.

For example, consider the following typical situation that exemplifies the scenario discussed above: suppose that we want to count the number of occurrences of each word contained in a set of documents. The number of words may get very large (say, millions), so we want to minimise object creation. Thus, while parsing each document we use a `StringBuffer` to accumulate characters and, once we've got our word, we would like to check whether it is in a dictionary. Since `StringBuffer` does not override `equals`, we have to make it into a `String` firstly. Now, this apparently innocuous action is really causing havoc: the new string will get the buffer backing array, and the buffer will mark itself as "shared". If the word just found must be added to the dictionary,

we will insert a string containing a character array potentially much longer than needed (as `StringBuffer`'s backing arrays grow exponentially fast). Even worse, the backing array of our `StringBuffer` cannot be reused, even if we did not really insert the word into the dictionary. In fact, it cannot be reused in any case, as a call to `setLength(0)` will reallocate it to a standard length.

The purpose of this paper is to describe a new string class, `MutableString`, which tries bridge the schism between `String` and `StringBuffer`. The name was chosen so to highlight the fact that we are aimed at replacing `String`, but we want to keep the mutable nature of `StringBuffer`. Some other proposals to replace the standard Java string classes have appeared in the last years, for instance [17,18].

It may be argued that in a lot of other situations `String` and `StringBuffer` are efficient. Nonetheless, in many power application the operations performed by these classes behind the scenes may be very harmful. The following is a simple benchmark counting the number of occurrences of words in a 200 Mbyte text file:[1]

	words/s (Linux)	words/s (Solaris 9)
`String`/`StringBuffer`	902843	283961
`MutableString`	2360994	580478

Note that we do not claim that it is not possible to work around the problem and use `String` and `StringBuffer` in a better way: the problem is that to do so you must take into account non-documented behaviours that are clear only to people knowing the API source code in depth.

`MutableString`, in contrast, has been designed so that its inner workings are extremely clear and well documented, trying to make all trade-offs between time and space explicit, and clarifying the number of new objects generated by a method call. This allows us to keep under control the hidden (and potentially very heavy) burden of garbage collection; since the latter runs in time proportional to the number of *alive objects*, incrementing the frequency of garbage collection has a cost that can grow independently of all other parameters. This aspect is usually overlooked in the choice for more efficient algorithms, but it often backfires (as the Java string classes show).[2]

One should remark that immutable strings are much safer than mutable strings. Indeed (for obvious reasons) *any* data structure becomes much safer when it is made immutable, especially if immutability is enforced by the language type-checking mechanisms. Nonetheless, we believe that immutable types are inappropriate for data structures that undergo massive manipulation (e.g., strings), as every modification leads to the creation, and eventually to the collection, of objects. This consideration, of course, may not be

[1] The benchmarks were produced on a Pentium 2.4 GHz running Linux, and on a Sun Fire V880 based on SPARC 900 MHz processors and running Solaris 9.

[2] Of course, garbage collection techniques become more and more sophisticated every day, and the scenario we have described may suddenly become unexpectedly "gc friendly". Moreover, there is a line of thought that advocates devising usage patterns that are expected to reduce the burden of garbage collection (e.g., object pooling). However, we strongly believe that trying to discipline program design around the current state of the art in garbage collection techniques is short sighted. More generally, it is better not to dirty your house than to buy more and more expensive cleaning tools.

appropriate for safety-critical applications, where you don't want to trade robustness for efficiency; it should be noted, however, that the interruptions caused by the garbage collector are, in fact, one of the main obstacles to the usage of Java in real-time applications, and most safety-critical software is also real time.

Efficiency is an important *leitmotiv* in the design of `MutableString`; thus, in the second part of the paper we turn to another typical dychotomy of Java string classes (but the same may be said of other object-oriented languages). If we want to perform search and replace operations on a string, we are offered just two ways of proceeding: either, in the case of a constant string, by using a very inefficient, brute-force double loop, or by setting up all the machinery that is necessary to perform a regular-expression search.

The choice of implementing searches in `String` using a brute-force double loop, as naive as it might seem, is not completely unjustified. Indeed, there are several sophisticated text-search algorithms available in the literature, but all of them require a significant set-up overhead, which makes them unsuitable for a general-purpose method. For example, the simplest implementation of the Boyer–Moore search algorithm [6] requires us to set up a vector of integers with $|A|$ elements, where A is the alphabet. This aspect is often overlooked, as it is considered a constant-space component of the algorithm. However, Java uses Unicode as native string alphabet, and reserving a table for 2^{16} characters, or, even worse, 2^{31} collation elements, is out of the question.

The result is that many common searches end up being either too slow (because of brute-force methods), or too resource consuming (because of the large number of objects that must be generated and collected for a regular-expression search).

The search methods implemented in `MutableString` use instead a kind of *soft data structure*, that is, they help the search using a simple, randomised, inexact data structure, a *compact approximator*, which has an extremely low set-up cost, but gives a significant advantages over the brute-force method.

Compact approximators are a generalisation of Bloom filters [4]; they were devised as a low-cost alternative to sophisticated data structures such as hash tables during the development of `MutableString`. Using compact approximators, we implement a few relaxed versions of Daniel Sunday's QuickSearch [16], a variant of the Boyer–Moore algorithm. Since compact approximators have a more general appeal, we will describe them in their full generality before explaining how we have applied them to the text-search problem.

`MutableString` is distributed as free software under the GNU Lesser General Public License within the MG4J project (http://mg4j.dsi.unimi.it/).

2. Design goals

String handling is one of the most application-dependent kinds of code, and trying to devise a string class that satisfies everybody may lead to a class satisfying nobody.

The design of `MutableString` acknowledges that there are really two kinds of string: dynamical, fast-growing, write-oriented strings and frozen, static (if not immutable), read-oriented strings. The radical departure from the standard string classes is that the two natures are incorporated in the same Java class.

The other design goal of `MutableString` is efficiency in space and time. We have in mind applications storing and manipulating dozens of millions of strings; transformations such as upcasing/downcasing, translations, replacements etc. should be made in place, benefit from exponentially growing backing arrays, and be implemented with efficient algorithms.

As far as memory occupation is concerned, we do not want to waste more than an integer attribute, beside the backing array (notice that a integer attribute is the minimum requirement if you plan to have exponentially growing backing arrays). On the other hand, we do not want to give up hash code caching (at least for "frozen" strings).

Finally, `MutableString` is a non-final class: it is open to specialisations that add domain-specific features; nonetheless, for maximum efficiency all fundamental methods are final.

3. Compactness and looseness

A mutable string may be in one of two modes, *compact* and *loose*. When in loose mode, it behaves more or less like a `StringBuffer`: its backing array is increased exponentially as needed, so that frequent insertion/append operations can be performed efficiently; when in compact mode, the backing array gets increased on request, as before, but no attempt is made to make it larger than it is strictly needed (the rationale being that if a compact string requires modifications they will be very rare: in this case, we prefer space occupancy to time performance). Moreover, when a loose `MutableString` is turned into a compact `MutableString`, the backing array has really the same length as its real content.

The `equals` method for `MutableString` is based on the string content, as in `String`, and the hash code is computed accordingly; the hash code of a compact `MutableString` is cached, although no attempt is made to recompute it upon changes (in that case, it simply becomes invalid).

Note that the mode has only influence on the expected space/time performance, not on the object semantics: a `MutableString` behaves exactly in the same manner, regardless of its mode, although changes are more expensive on compact strings, and the computation of hash code is more expensive on loose strings.

`MutableString` provides two explicit methods to change mode; all remaining methods (except `ensureCapacity`) preserve the string mode, and there are two methods to test whether a `MutableString` is in a given mode or not. A mutable string created by the empty constructor or the constructor specifying a capacity is loose; all other constructors create compact mutable strings.

4. Implementation choices

Attributes. A mutable string contains two attributes only: `array`, a backing character array that contains the actual string characters, and `hashLength`, an integer value that embodies information about the mode of the string, and its length or its hash code.

More precisely, if `hashLength` is negative, the string is compact, and corresponds to the entire content of the backing array; moreover, the value of `hashLength` is the hash code of the string (-1 represents an invalid hash code). Otherwise, the string is loose, and

`hashLength` represents the actual length of the string, that is, the valid prefix of the backing array. All in all:

(1) `hashLength` ≥ 0: the string is loose, and `hashLength` contains its length;
(2) `hashLength` $= -1$: the string is compact, and coincides with the content of `array`, but its hash code is unknown;
(3) `hashLength` < -1: as above, but `hashLength` contains the hash code.

The hash code of a mutable string is defined to be the hash code of the content-equivalent string with the highest bit set. Note that in this way the empty string has a valid hash code.

Reallocations. Backing array reallocations use a heuristic based on looseness. Whenever a new capacity is required (because of an insert or append operation), compact strings are resized to fit *exactly* the new content. In contrast, the capacity of a loose string needing new space is computed by maximising the new length with twice the current capacity.

The effect of this policy is that reused strings or strings created without an initial content will get large buffers quickly, but strings created with other constructors and with few changes will occupy little space and perform very well in data structures using hash codes.

Thus, reused or otherwise heavily manipulated strings may have a rapid growth, if needed, and when their state is not to change anymore you can compact them (of course, compacting a string may require reallocating the backing array).

Exposing internals. It is well known that encapsulation and information hiding are essential in object-oriented systems. Nonetheless, in our opinion `String` and `StringBuffer` are a bit too drastic in forbidding any access to their backing arrays and internal variables. `MutableString`, in contrast, does not inhibit subclasses to manipulate `array` and `hashLength`. In fact, you can even get (at your own risk) a direct reference to `array`; since hash codes are cached, the `changed()` method should be invoked immediately after modifying the backing array so to force an invalidation of the cached value. Note, however, that is not possible, even with the access provided, to make the object state incoherent up to the point of causing an exception to be thrown: the only mismatch that may happen is between a compact string content and its cached hash code.

5. Method optimisation: searching

One of the fastest known algorithms for searching a pattern (i.e., a string) in large texts is the Boyer–Moore algorithm [6], along with its many variants (e.g., [1,7,8,14,16]). All variants are based on the following simple idea: Let p be the pattern of length P and t be the text of length T to be searched.[3] The pattern occurs in position k if $p_i = t_{k+i}$ for all $0 \leq i < P$. Now, when examining a candidate position k, we compare the characters t_{k+m-1}, t_{k+m-2}, and so on, with the corresponding characters of the pattern. If a mismatch is detected at some point, we have to increment k, and consider a new candidate position. To choose the increment for k (called the *shift*), we can exploit many heuristics based on information obtained during the scan, and on the structure of the pattern.

[3] We will write p_i and t_i for the i-th character of p and t, starting from 0; the characters are drawn from a fixed alphabet A.

A well known heuristic used in the basic version of the algorithm (and in almost all variants) is the so-called *bad-character shift*. Suppose that while examining position k we find a mismatch at index j, that is, $p_j \neq t_{k+j}$ but $p_h = t_{k+h}$ for $h > j$. Instead of incrementing k by just one, we may want to align the pattern so that the last occurrence of t_{k+j} in the pattern is in position $k + j$, at least if the last occurrence of t_{k+j} happens before position j. Note that if t_{k+j} does not occur at all in the pattern we can directly shift the pattern by $j + 1$.

For instance, if we have a mismatch on the first check, that is, in position $P - 1$, and t_{k+P-1} does not appear in the pattern we can shift the pattern by P (albeit apparently infrequent, on large alphabets and small patterns this case occurs fairly often). This will clearly speed up the search significantly.

Thus, in general, the shift is computed as $j - \ell(t_{k+j})$, where $\ell(c)$ denotes the index of the last occurrence of character c in the pattern, or -1 if the character does not appear. If the resulting value is not positive, the pattern is shifted by one position.

The key ingredient for implementing the bad-character shift heuristic is a *shift table* that stores $\ell(c)$, and this is indeed how this heuristic is usually implemented in most text-search libraries. In Java, however, using a shift table is out of the question, because of the alphabet size.[4]

Obvious solutions come to mind: for instance, storing this information in a hash table. However, this approach raises still more questions: unless one is ready to handle rehashing, it is difficult to estimate the right table size, as it depends on the number of *distinct* characters in the string (even an approximate evaluation would not completely avoid the need for rehashing). Moreover, the table should contain not only the shifts, but also the keys, that is, the characters, and this would result in a major increase of space occupancy. Finally, the preprocessing phase could have a severe impact on the behaviour of the algorithm, in particular on short texts. These considerations hold a fortiori for more sophisticated data structures, such as balanced binary trees.

A very simple solution to this problem has been proposed in [9]. Observe that we can content ourselves to store upper bounds for $\ell(c)$: this could slow down the search, but certainly will not produce incorrect results. Thus, instead of using a hash table with standard collision resolution, one might simply use a fixed-size table and combine colliding values using maximisation.

This approach, however, has several drawbacks: first of all, it does not allow trade-offs between space and errors; second, the birthday paradox makes it easy to get collisions, even with relatively large tables; third, the technique is patented, and thus cannot be freely used in academic or open source work.

We solve these problems by using the good statistical properties of *compact approximators*, a generalisation of Bloom filters [4], a technique from the early 70s that has seen recently a revival because of its usefulness in web proxies (see, e.g., [10]). Starting from an approximation of the number of distinct characters in the pattern, we provide a way to store an upper bound to ℓ that is tunable so to obtain a desired error

[4] We shall see, in fact, that even if one agrees to allocate a large amount of memory for a shift table, the results are much worse than expected.

probability; with respect to a hash table, one of the main advantages is that a bad estimate for the number of distinct characters can make the approximation worse, but it is otherwise handled gracefully.

To be as general as possible, in the remaining part of this section we will discuss our solution casting it into the more general framework of monotonic approximation of lattice functions.

5.1. Compact approximators

For each natural number $n \in \mathbf{N}$, we denote with $[n]$ the set $\{0, 1, \ldots, n-1\}$.

Let L be a lattice [3], whose partial order relation is denoted by $\leq$; the greatest lower bound (least upper bound, respectively) of two elements $x, y \in L$ will be denoted by $x \wedge y$ ($x \vee y$, resp.). L is assumed to contain a least element, denoted by $\bot$ (bottom).

Let Ω be a fixed set. Our purpose is to provide a data structure to represent, or to approximate, functions $\Omega \to L$. Let $f : \Omega \to L$ be a function; its *support* $D(f)$ is the set of all elements of the universe that are not mapped to $\bot$, that is,

$$D(f) = \{ x \in \Omega \mid f(x) \neq \bot \}.$$

In our application, of course, Ω is the alphabet and L is the lattice of natural numbers.

Definition 1. Let $d > 0$ and $m > 0$. A *d-dimensional m-bucket compact approximator* for functions from Ω to L is given by a sequence of d independent hash functions $h_0, h_1, \ldots, h_{d-1}$ from Ω to $[m]$, and by a vector b of m values of L. The elements of b are called *buckets*.

When using an approximator to store a given function $f : \Omega \to L$, we fill the vector b as follows:

$$b_i = \bigvee_{\exists j < d \; h_j(x) = i} f(x).$$

In other words, the vector b is initially filled with $\bot$. For each $x \in \Omega$ such that $f(x) \neq \bot$, a d-dimensional approximator spreads the value of $f(x)$ in the buckets of indices $h_0(x), h_1(x), \ldots, h_{d-1}(x)$; when conflicts arise, the approximator stores the maximum of all colliding values.

Now, the function *induced* by the thus filled approximator is defined by

$$\tilde{f}(x) = \bigwedge_{j < d} b_{h_j(x)}.$$

In other words, when reading $f(x)$ from an approximator we look into the places where we previously spread the value and compute the minimum.

The interest of approximators lies in the following (obvious) property:

Theorem 1. *For all $x \in \Omega$, $f(x) \leq \tilde{f}(x)$.*

Note that in the case of the boolean lattice $L = \{0, 1\}$ we obtain exactly a Bloom filter (by approximating the characteristic function of a subset of Ω), whereas in the case $d = 1$ we obtain the structure described in [9].

As an example, consider the function $f : \Omega = [12] \to \mathbf{N}$ given by $f(1) = 3$, $f(5) = 1$, $f(9) = 2$ and $f(x) = 0$ in all other cases. We let $d = 2$, $b = 6$, $h_0(x) = \lfloor x/2 \rfloor$ and $h_1(x) = 5x \bmod 6$ (these functions have been chosen for exemplification only). In the upper part of Fig. 1, one can see how values are mapped and maximised into the buckets; in the lower part, one can see how some of the values are extracted. Note that some values are obtained from a single bucket, because h_0 and h_1 coincide. The values for which $\tilde{f}(x) > f(x)$ are highlighted.

Our interest is now in estimating how often $\tilde{f}(x) = f(x)$. We divide our evaluation into two parts: the bottom case and the nonbottom case.

The bottom case. Since we have in mind functions with a small support, we should look carefully at

$$\phi = \Pr[\tilde{f}(x) \neq f(x) \mid f(x) = \bot],$$

that is, the probability of erroneous computation at a point in which $f(x) = \bot$. The analysis is similar to that of a Bloom filter: if $|D(f)| = n$ (i.e., f is non-$\bot$ in n points), the probability that a bucket contains $\bot$ is

$$\left(1 - \frac{1}{m}\right)^{dn}.$$

To compute the wrong value, we must find non-$\bot$ values in all the buckets over which we minimise. This happens with probability

$$\phi = \left(1 - \left(1 - \frac{1}{m}\right)^{dn}\right)^{d} \approx \left(1 - e^{-\frac{dn}{m}}\right)^{d}. \tag{1}$$

The expression on the right is minimised at $d = m \ln 2/n$; the minimum is then $(1/2)^d$. This is a very good approximation of the exact result, as long as m is sufficiently large: Fig. 2 shows the relative error in the choice of d, comparing the integer closest to $m \ln 2/n$ and the integer minimizing the left-hand side of (1); note that even if the relative error may seem to grow significantly when n is smaller than m, the absolute error is always at most one.

The estimates above show that we can reduce exponentially the error probability by using more hash functions and a larger vector; the vector should be sized approximately as $m = 1.44\, dn$.

Note that the presence of multiple hash functions is essential: for instance, when $d = 3$ and $m = 3n/\ln 2$ we have $\phi = 1/8$, whereas a single hash function gives $\phi \approx 1/5$.

More precisely, if we set $d = 1$, since

$$1 - e^{-\frac{n}{m}} = \frac{n}{m} + O\left(\frac{n^2}{m^2}\right)$$

when the ratio m/n grows we get inverse linear error decay, as opposed to exponential.

The nonbottom case. This is definitely more complicated. Our interest is now in estimating

$$\psi = \Pr[\tilde{f}(x) \neq f(x) \mid f(x) \neq \bot],$$

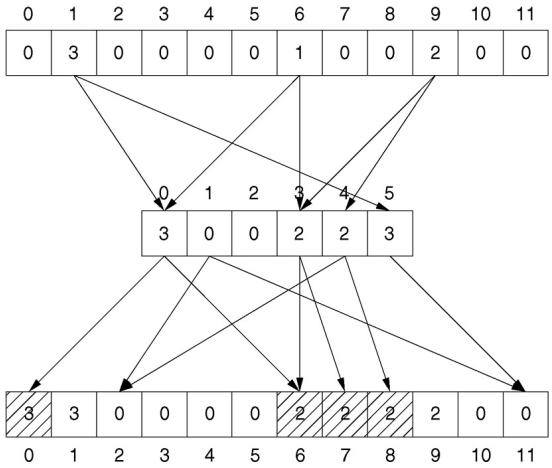

Fig. 1. A diagram representing a compact approximator.

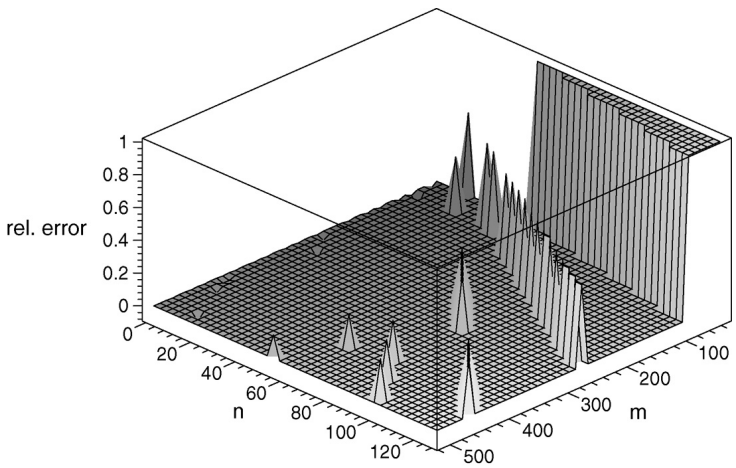

Fig. 2. A plot of the relative error produced by approximation (1).

that is, the probability of erroneous computation at a point in which $f(x) \neq \perp$. Note that

$$\psi = \sum_{i=0}^{s} \Pr[\tilde{f}(x) \neq f(x) \mid f(x) = v_i] \Pr[f(x) = v_i \mid f(x) \neq \perp]$$
$$= \sum_{i=1}^{s} \frac{a_i}{n} \Pr[\tilde{f}(x) \neq f(x) \mid f(x) = v_i].$$

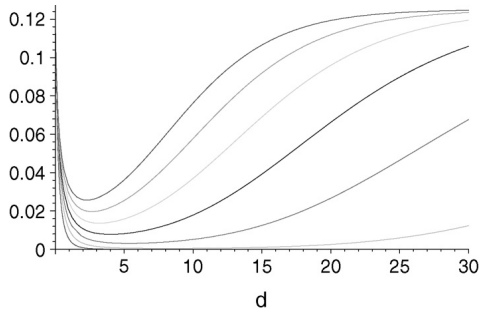

Fig. 3. The error probability of each summand in the uniform case, with $n = 8$ and $m = 2n/\ln 2$.

We now need to make some assumptions on the distribution of the values assumed by f. Suppose that f assumes values $\perp = v_0 < v_1 < \cdots < v_s$, and that it assumes value v_i exactly $a_i > 0$ times (i.e., $|f^{-1}(v_i)| = a_i$). Then,

$$\Pr[\tilde{f}(x) \neq f(x) \mid f(x) = v_i] = \left(1 - \left(1 - \frac{1}{m}\right)^{d\sum_{j=i+1}^{s} a_j}\right)^d$$

since the event $\tilde{f}(x) \neq f(x)$ takes place iff each of the d buckets assigned to x is occupied by at least one of the $\sum_{j=i+1}^{s} a_j$ elements with values greater than v_i. All in all we get

$$\psi = \sum_{i=1}^{s} \frac{a_i}{n}\left(1 - \left(1 - \frac{1}{m}\right)^{d\sum_{j=i+1}^{s} a_j}\right)^d.$$

The previous summation is not going to be very manageable. As a first try, we assume that f takes each value exactly once (as in our main application), getting to

$$\psi = \frac{1}{n}\sum_{i=1}^{n-1}\left(1 - \left(1 - \frac{1}{m}\right)^{d(n-i)}\right)^d,$$

where we used the fact that for $i = s$ the summand is always zero (there is no way to store erroneously the maximum value attained by f) and that now $s = n$. Note that for each summand we can apply the usual approximation

$$\left(1 - \left(1 - \frac{1}{m}\right)^{d(n-i)}\right)^d \approx \left(1 - e^{-\frac{d(n-i)}{m}}\right)^d.$$

Thus, each summand (which gives n times the error probability for value v_i) is minimised by $d = m\ln 2/(n-i)$. This is very reasonable: larger values gain from numerous hash functions, as they are likely to override smaller values. For the smallest value, the probability of error is very close to that of $\perp$. The situation is clearly depicted in Fig. 3, which shows the error probability as a function of d.

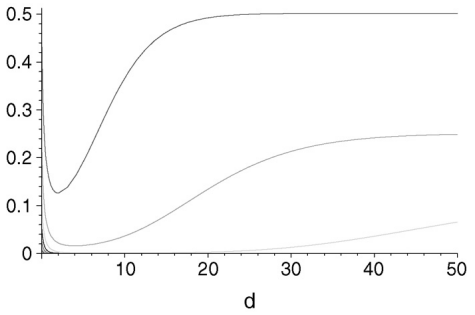

Fig. 4. The error probability of each summand in the exponential case with $n = 255$ and $m = 2n/\ln 2$.

Even if we cannot provide an analytical minimum for ψ, the assumption that $D(f)$ is very small w.r.t. Ω makes a choice of d that minimises ϕ sensible.

Exponential distribution. A similar partial analysis can be carried out if the values are distributed exponentially, that is, $v_i = 2^{(s-i)}$; in particular, this means that $f(x) = \perp$ on half of Ω. In this case, $\sum_{j=i+1}^{s} a_j = 2^{s-j+1} - 1$, $n = 2^{s+1} - 1$ and we get

$$\psi = \sum_{i=0}^{s-1} \frac{2^{s-i}}{n} \left(1 - \left(1 - \frac{1}{m}\right)^{d2^{s-i+1}}\right)^d,$$

where this estimate includes (when $i = 0$) the contribution of the bottom case.

In this scenario it is even more sensible to tune the choice of d and m/n using the bottom case. Indeed, all summands behave much better (on one hand, the error has a lesser impact as i grows, as v_i decreases exponentially; on the other hand, larger values have a greater probability of being stored exactly), as shown in Fig. 4.

5.2. Using approximators in the Boyer–Moore algorithm

The previous discussion paves our way toward an implementation of the Boyer–Moore algorithm that uses an approximator to store the bad-character shift table. Recall that the function we want to approximate is $\ell : A \to \mathbf{Z}$, where $\ell(c)$ is the index of the last occurrence of c in the pattern, or -1 if the character does not occur. For sake of implementation efficiency, we will indeed approximate $\ell' : A \to \mathbf{N}$, with $\ell'(c) = \ell(c) + 1$, so that $\perp = 0$.

Notice that having an upper bound for ℓ' is sufficient for the Boyer–Moore algorithm to work correctly, because of the way shifts are computed. More precisely, when analysing a given candidate position k in the text, if the j-th character of the pattern is the rightmost mismatch, and c is the text character found in that position (the *bad character*), then we compute the shift as $\max(1, j - \ell'(c) + 1)$. Having a larger value for ℓ' has the simple effect of reducing the shift. This is true even for variations of the algorithm that look at different characters to compute the shift (e.g., [16]).

A very noteworthy feature of compact approximators is that, unlike exact data structures (e.g., hash tables), their memory footprint may be limited a priori. Suppose that we are

confronted with a pattern containing, say, 100 000 distinct characters: if we want to keep an exact bad-character shift table, we have to build a very large data structure—there is no way to trade some memory for speed. In contrast, a compact approximator may be arbitrarily bounded in size: the effect of the size bound is simply to make the approximation worse.

In particular, if we decide to bound the size of an approximator so that it fits into the processor's cache, the loss in precision is likely to be largely compensated by faster access: see, in particular, the paradoxical results obtained on the Pentium and shown in Table 4, where using the simplest exact data structure—an array—gives actually *much worse* timings than a compact approximator.

5.3. Implementation issues

Suppose we want to find occurrences of pattern p in text t. Devising an approximator for ℓ' requires choosing the various parameters involved in the approximation.

Estimating n. As a first step, we have to evaluate the number n of distinct characters in p; a rough estimate is given by the length of p, but you can try to adopt more sophisticate techniques to get a better bound for n (see, e.g., [2,11]). Note that these are constant-space, linear-time techniques that give just an approximation, but this is perfectly acceptable, as approximation errors lead only to better or worse precision in representing ℓ' (depending on whether the error is on the upper or lower side).

Choosing d and m. Then, we must decide the number m of buckets and d of hash functions we are going to use for the approximator. As explained above, one should choose m and d so that $d \approx \ln 2\, m/n$.

According to the analysis outlined in the previous sections, a larger value for d reduces the error probability; on the other hand, choosing a value of d that is too large may severely reduce the performance, because both the memory requirement (number of buckets) and the time needed to consult the approximator grow linearly with d.

The choice of d and m, hence, depends subtly on the quantity of memory available, and on the trade-off between the time one needs to compute the d hash values and the time that is wasted when a short skip is computed as a consequence of the imprecise evaluation of ℓ'. It is also a good idea to maximise the computed value of m with a reasonable constant, so to compensate the set-up overhead on very short patterns.

Experimental results. We ran a number of experiments to determine the performance of our solution. The most important question concerns the number of positions that are considered for matching; in particular, we are interested in the ratio between the number c_{app} of candidate matching positions considered by our approximate algorithm w.r.t. the optimal number c of candidate positions considered by an exact implementation of the Boyer–Moore algorithm using an entire skip table (in both cases, we are only adopting the bad-character heuristic).

Fig. 5 shows the ratios c_{app}/c as a function of d for various situations. The diagram on the left shows the ratios in the case of three patterns of lengths 9, 18 and 27 made of characters that appear frequently in the text. Clearly, the relative performance of the approximate algorithm decays as the pattern length grows. The diagram on the right shows the same data for patterns made of characters that appear rarely in the text.

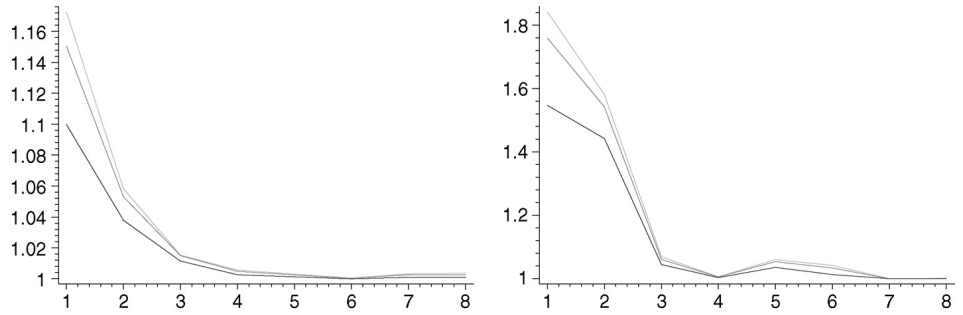

Fig. 5. Ratios c_{app}/c for frequent and rare patterns, with varying d.

We remark that in the first case two hash functions are sufficient to limit the increment of c_{app} to about 6%, even for the longest pattern. In contrast, in the second case we need three functions to get similar results. The reason is clear: in the latter case the cost of imprecise representation is high, because almost all characters in the text to be searched should be $\perp$ valued (and thus provide a large skip), but this will not happen with a too imprecise representation. The loss is reduced in the former case because some of the most frequent characters are not $\perp$ valued. In general, the evaluation of the effectiveness of a compact approximator should be based on a distribution of the inputs that it will process.

The data we gathered suggest that three hash functions with $4.3n$ buckets are a good choice; also two hash functions with $2.9n$ buckets behave reasonably, but have a sensible loss with rare patterns. Note the strange peak at $d = 5$ for a rare pattern. This phenomenon is almost unavoidable for some values of d, and it is due to the fact that increasing the number d of hash functions may cause (in a transient way) an error in the computation of ℓ' for a very frequent text character, such as "e" in English. The penalty in this case is very high, because comparisons with that character will be frequent, and will increase substantially the number of positions considered for matching.

Note that for "on-the-fly" searches, in particular on short strings, even setting up a compact approximator could be too much. However strange it may seem, we can still use a compact approximator—in fact, its boolean form, that is, a Bloom filter—to obtain a very lightweight data structure that involves *no object creation at all*: a single 32- or 64-bit integer can be used to fit a one- or two-hash boolean approximator. The approximator records approximately which characters appear in the pattern. Thus, skipping is performed using the entire pattern length, or it is not performed at all. This technique generates no objects, but speeds up significantly searches, even for small patterns in small texts.

6. Method optimisation: multi-character searches and replacements

Multi-character search and replace operations are very common in text processing, and in particular in web applications. They usually take the form either of locating the first

occurrence of a character in (or out of) a certain set, or of computing the maximal prefix of a string made of characters in (or out of) a certain set.

For instance, a very common usage pattern for strings in web applications is *escaping*: to be passed to some interpreter fragments of text, it is necessary to replace characters in a certain set, transforming them in a suitable form that will be detected by the interpreter (the most typical usage involves replacing the characters `<`, `>`, `&quot;` and `&` with the corresponding SGML entities).

Escaping cannot be performed (for obvious reasons) one character at a time: again, we are left with setting up a complex (and slow) regular expression that will substitute each character with its translation.

In contrast, `MutableString` offers a family of `replace` methods, which allow one to substitute all occurrences of characters from a given set with a corresponding `String` (character, `CharSequence` etc.). In particular, we implement a version in which the set of characters is specified by an array, obtaining nonetheless a low set-up time and very fast searches (in fact, in some cases even faster than the method taking a collection of characters as argument). Not surprisingly, we base again our implementation on a Bloom filter.

Consider, for example, the method invocation `s.replace(ca, sa)` where `s` is a `MutableString`, `ca` is a character array and `sa` is an array of `String`s. This method should substitute every occurrence of `ca[i]` with `sa[i]` for all indices `i` less than the length[5] of `ca`.

The `replace` method must scan `s` twice: the first scan is needed to compute the length of the new string, whereas the second scan actually performs the substitutions. In both scans, when analysing a certain character `s[k]`, we should check whether the character should be replaced, in which case we should also determine what is the string to be substituted. This would require a linear scan of `ca`, and most of these steps will probably end up scanning the whole array before knowing that no substitution was really needed (i.e., that there was no `i` such that `s[k]==ca[i]`): in fact, we expect that only a small fraction of characters in the string requires a replacement.

A straightforward solution to this problem is that of creating from `ca` and `sa` a `Map` (from `Character`s to `String`s) and then using the `get` method instead of scanning the original array. From a theoretical viewpoint, this solution might largely reduce the time needed to test whether a character requires a substitution (the `get` method requires logarithmic time in the case of a `TreeMap`, and constant expected time in the case of a `HashMap`, under the assumption that the hash codes are evenly spread). Unfortunately, this solution requires the creation of a `Map` object and of as many `Character` objects as the length of `ca`; moreover, the cost of a method call for each test should be taken into account.

As an alternative, one could try to use a home-made tiny hash table to store the set of characters to be substituted, but dimensioning it becomes problematic: hash tables are not well suited for situations, like this, where almost all tests are expected to give a negative answer. Neither is it possible to use a table storing, for each character, the index where it appears in the array, if there is one, because such a table would be exceedingly large for Unicode.

[5] The arrays `ca` and `sa` must have the same length; moreover, it is assumed that no character appears more than once in the array `ca`.

The solution adopted in `MutableString` is to use a one-hash Bloom filter to represent in an approximate way the set of characters to be searched for. Then, every time a character `s[k]` is examined, we first check whether it is rejected by the filter: if so, then the character does not need to be substituted. Otherwise, we have to scan the array.

Of course, false positives are possible, but they are quite rare: more precisely, with rough but reasonable probabilistic hypotheses on the inputs and the set of characters to be replaced, and assuming that most characters need no replacement, the gain in speed is easily calculated using the formulae of [4], and turns out to be

$$\frac{m^n}{m^n - (m-1)^n},$$

where m is the number of bits in the filter and n the length of `ca` (omitting, however, the overhead that is necessary to check the content of the filter). In particular, using $m = 32$ we have an eightfold speedup using four characters. The speedup is still more than twofold with 20 characters. Of course, as 64-bit processors become more common, it may be reasonable to use a 64-bit mask. Benchmarks confirmed that this approach is very effective: indeed, it very often outperforms a map-based implementation, because it does not invoke any method.

In our current implementation, we hash characters using simply their least significant bits: this solution does not require any method call and can be computed in an extremely optimised way; moreover, because of the way Unicode charts are organised, we expect that natural-language documents contain characters that differ only in their least-significant bits.

7. Benchmarking

Text search. We present benchmarks of the implementation of compact approximators used in `MutableString`.[6] The benchmarks were produced on a Pentium 2.4 GHz running Linux, and on a Sun Fire V880 based on SPARC 900 MHz processors and running Solaris 9, using in both cases the Sun 1.4.1 JDK.

Benchmarking Java code is not an easy task, as the virtual machine performs several activities that may slow down (garbage collection) or speed up (just-in-time compilation) the execution. Our test were performed a large number of times, discarding the first results (as they are slower than necessary, due to the virtual-machine warm-up) and averaging over several executions.

During the development of our classes, we conducted extensive benchmarking, in particular to tune finely the inner loops. Among the tests we performed, we selected a small sample of cases that, we believe, give a flavour of the kind of speed-up provided by our techniques. We compare four pattern-search methods:

[6] More precisely, the implementation of approximators is contained in a separate class, `TextPattern`, used by `MutableString`. It should be noted that the current distribution features several tweaks to fine-tune the techniques presented in this paper: for instance, US-ASCII characters are treated separately using a vector, as they appear frequently in almost every Unicode text. The benchmarks we are reporting, however, are obtained using a direct implementation of the algorithms in the form they are discussed in this paper.

- a brute-force double loop (as implemented in the `indexOf()` method of `String`);
- an exact implementation of the Boyer–Moore algorithm (more precisely, of its variant known as QuickSearch [16]), using a Java `Map` to store the bad-character shift table;[7]
- another exact implementation of QuickSearch, using an array;
- an approximate implementation of the same algorithm, using compact approximators and one of the approximate counting algorithms described in [2].

Our tests were performed on two 16 Mbyte documents (one produced at random, and the other one containing a US-ASCII English text), and consisted in searching (all occurrences of) a nine-character and a 54-character pattern. In an attempt to account for garbage-collection overhead, we additionally provide timings obtained by taking into account also the time required by a call to the `System.gc()` method, which suggests the virtual machine to perform garbage collection. All timings are in milliseconds.

The reader will notice that compact approximators work in all cases much better than maps and better than the brute-force approach (even though, of course, on very short patterns a brute-force loop will outperform any sophisticated algorithm). The timings for arrays are given just for comparison, because, as we have already remarked, it is usually not practical to allocate such a large array (a quarter of megabyte in the case of Unicode).

It is interesting to note that, in the case of a long pattern on a random text, the very sparse memory accesses of the array implementation makes it even *slower* than the approximator-based one, as most memory accesses are cache misses.

A final *caveat*: the impact of garbage collection may seem small, but the reader must take into consideration that *almost no objects were alive during the collection*. As we mentioned in the introduction, in a real-world large applications, the collection time may be much larger, even when searching the same pattern within the same text.

Table 1
English 16 Mbyte text, nine-character pattern

	Pentium (Linux)		SPARC (Solaris)	
	w/ gc	w/o gc	w/ gc	w/o gc
brute force	60		335	
`Map`	120	114	402	395
array	30	27	190	185
approximator	56	52	252	238

Table 2
English 16 Mbyte text, 54-character pattern

	Pentium (Linux)		SPARC (Solaris)	
	w/ gc	w/o gc	w/ gc	w/o gc
brute force	55		324	
`Map`	71	65	224	216
array	26	22	113	111
approximator	31	28	134	131

[7] Since Java does not provide maps handling primitive types without wrappers, we really used a type-specific hash-table map from `fastutil` (http://fastutil.dsi.unimi.it/).

Table 3
Random 16 Mbyte text, nine-character pattern

	Pentium (Linux)		SPARC (Solaris)	
	w/ gc	w/o gc	w/ gc	w/o gc
brute force	50		306	
`Map`	116	108	365	359
array	49	45	189	181
approximator	48	42	200	194

HTML-ising a text. Just to give another hint of the performance of `MutableString`, we consider a simple task: HTML-ising a string. We start from a web page of about 100K, and iteratively replace all occurrences of `&` with `&`.

Table 4
Random 16 Mbyte text, 54-character pattern

	Pentium (Linux)		SPARC (Solaris)	
	w/ gc	w/o gc	w/ gc	w/o gc
brute force	49		310	
`Map`	47	40	144	136
array	37	33	85	83
approximator	25	22	92	87

Type	calls/s (Linux)	calls/s (Solaris 9)
compact `MutableString`	145.56	141.64
loose `MutableString`	581.39	199.20
`StringBuffer`	36.57	12.38
unsync'd `StringBuffer`	37.16	12.64
`String`	109.41	74.12

It should be said that the test had to be run on `StringBuffer` using an external loop calling repeatedly `lastIndexOf()` and `replace()`, and that the test on `String` used regular expressions (neither class contains something corresponding to the versatile `replace()` method of `MutableString`). Note also that this test was run without causing garbage collection: the reader should thus consider the result obtained by `String` as a bit optimistic. The unsynchronised buffer case was obtained by recompiling `StringBuffer` after stripping all `synchronize` keywords.

The `length()` method. Of course, no class can be both more compact and faster: space and time have their own laws and trade-offs. For instance, the `length()` method has to check whether the string is compact or loose, and act accordingly. The following benchmark gives an idea of the relative loss:

Type	Mcalls/s (Linux)	Mcalls/s (Solaris 9)
compact `MutableString`	285	89
loose `MutableString`	359	45
`StringBuffer`	64	8
unsync'd `StringBuffer`	393	223
`String`	393	177

It should be noted that on such a short method the results are mostly dependent on architectural issues (caches, method inlining, etc.).

8. A string–freedom manifesto

In general, Java is a very flexible, well designed, complete language, with vast and carefully structured APIs; its mind-boggling complexity pays only a small price to space/time efficiency, which makes it more and more appealing for the development of critical, large applications.

APIs are generally designed so that you can simply rewrite a class (or a bunch of classes) if you are not satisfied with its performances; so, a typical programming pattern consists in using the standard, general-purpose APIs in the first steps of development, and then, perhaps after profiling, in substituting only those classes that are critical with other, hand-tailored versions.

Of course, you have to pay a price for this. For example, suppose that `java.net.URL` performs too badly for your needs, and you want to change it with your version `foo.bar.MyURL`. You are free to do so, but what about all classes and methods in the `java.net` package that rely on `URL`? If you use them, you will probably have to rewrite more classes: there is no way out of this, since there is nothing like a URL-interface, and `URL` is `final`.

As you can expect, this price is small if you substitute, so to say, some exotic well hidden class of the hierarchy, but it becomes high if you substitute some fundamental, pervasive class, like `String`.

So what is the price you have to pay if you want to get rid of `String`, substituting it with something else, say with `MutableString`? Of course, you can forget about string literals: every time you want to initialise a `MutableString` using a literal you have to use a `String` literal and throw it away immediately; however, there is little harm in doing so, except that the Java `String`-literal pool becomes virtually useless.

The use of the concatenation operator + becomes a trap: every time a `MutableString` is concatenated with +, it is first turned into a `String`. Finally, all I/O related methods accepting `String`s require an implicit or explicit call of `toString()`.

`MutableString` tries to lessen this burden by providing built-in methods for common I/O operations: for instance, you can use `s.println(System.out)` to print a `MutableString` to standard output.

A more reasonable solution, however, would be provided by a pervasive use in the Java core APIs of the new `CharSequence` interface. Character sequences are an abstraction of a read-only string, and are used, for instance, by the new regular expression facilities (indeed, you can split a `MutableString` on a regular expression *without creating a `String` object*, since `MutableString` implements `CharSequence`).

Every time there is a method accepting a `String`, there should also be a polymorphic version accepting a character sequence (`String` implements `CharSequence`, so to be true you do not really need two methods; however, calls through an interface are slower). Moreover, the string concatenation operator + should avoid useless calls to `toString` for objects which implement `CharSequence`. These small changes would actually make

a customisation of `String` and `StringBuffer` possible. As an additional optimisation, it would be very useful if `CharSequence` required the implementation of a `getChars()` method similar to that of `String`: in this way, bulk copies would be performed much more quickly.

Nonetheless, one has to remark that the main problem remains that `CharSequence` *specifies no contract for equality*. As a result, two classes implementing `CharSequence` may contain identical sequences of characters, and nonetheless they may end up not being equal (w.r.t. `equals()`). The pernicious side-effect is that it is impossible to mix instances of classes implementing `CharSequence` in data structures (e.g., hash tables). Moreover, it is not possible to interrogate containers filled with instances of classes implementing `CharSequence` (e.g., `MutableString`) using a constant `String`: for practical reasons, `MutableString` implements equality so that comparison with character-by-character equal strings will return true, but the same does not happen with `String`. There is, unfortunately, no easy way out, as both `String` and `StringBuffer` implement `CharSequence`, and they have different equality contracts.

9. Conclusions

We have presented `MutableString`, a new string class devised to make large-scale text processing easier and more efficient in Java. Mutable strings are extremely compact, and can behave more like `StringBuffer` or `String`, as needed. Moreover, they provide low-cost set-up, efficient search and replace methods based on a new approximated data structure. Many of the ideas presented in this paper, of course, are applicable to many other object-oriented languages as well (in particular, the usage of compact approximators for Boyer–Moore type algorithms on large alphabets).

References

[1] A. Apostolico, R. Giancarlo, The Boyer–Moore–Galil string searching strategies revisited, SIAM Journal of Computing 15 (1) (1986) 98–105.
[2] Z. Bar-Yossef, T.S. Jayram, R. Kumar, D. Sivakumar, L. Trevisan, Counting distinct elements in a data stream, in: Proc. 6th Internat. Workshop on Randomization and Approximation Techniques, September 13–15, 2002, Lecture Notes in Computer Science, vol. 2483, 2002, pp. 1–10.
[3] G. Birkhoff, Lattice Theory, third edn. (new), AMS Colloquium Publications, vol. XXV, American Mathematical Society, 1970.
[4] B.H. Bloom, Space-time trade-offs in hash coding with allowable errors, Communications of the ACM 13 (7) (1970) 422–426.
[5] P. Boldi, B. Codenotti, M. Santini, S. Vigna, UbiCrawler: a scalable fully distributed web crawler, Software: Practice & Experience 34 (8) (2004) 711–726.
[6] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Communications of the ACM 20 (10) (1977) 762–772.
[7] L. Colussi, Fastest pattern matching in strings, Journal of Algorithms 16 (2) (1994) 163–189.
[8] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter, Speeding up two string-matching algorithms, Algorithmica 12 (4–5) (1994) 247–267.
[9] M.E. Davis, Forward and reverse Boyer–Moore string searching of multilingual text having a defined collation order, U.S. Patent 5,440,482, assigned on August 8, 1995 to Taligent, Inc.
[10] L. Fan, P. Cao, J. Almeida, A.Z. Broder, Summary cache: a scalable wide-area Web cache sharing protocol, IEEE/ACM Transactions on Networking 8 (3) (2000) 281–293.

[11] P. Flajolet, G.N. Martin, Probabilistic counting algorithms for data base applications, Journal of Computer and System Sciences 31 (2) (1985) 182–209.
[12] A. Heydon, M. Najork, Mercator: a scalable, extensible web crawler, World Wide Web (1999) 219–229.
[13] A. Heydon, M. Najork, Performance limitations of the Java core libraries, Concurrency: Practice & Experience 12 (6) (2000) 363–373.
[14] T. Lecroq, A variation on the Boyer–Moore algorithm, Theoretical Computer Science 92 (1992) 119–144.
[15] N. Nagarajayya, J.S. Mayer, Improving java application performance and scalability by reducing garbage collection times and sizing memory using JDK 1.4.1, http://wireless.java.sun.com/midp/articles/garbagecollection2/.
[16] D.M. Sunday, A very fast substring search algorithm, Communications of the ACM 33 (8) (1990) 132–142.
[17] T.A.S. Foundation, `FastStringBuffer`, this is a class of Xalan-J, http://xml.apache.org/xalan-j/.
[18] T. Wang, 65% faster Java™ string buffer implementation, October, 2001, Technical Paper on http://www.hp.com/.