A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory*

Andreas Crauser
Max-Planck-Institut für Informatik
Saarbrücken, Germany
crauser@mpi-sb.mpg.de

Paolo Ferragina
Dipartimento di Informatica
Università di Pisa, Italy
ferragin@di.unipi.it

Abstract

The construction of full-text indexes on very large text collections is nowadays a hot problem. The suffix array [Manber-Myers, 1993] is one of the most attractive full-text indexing data structures due to its simplicity, space efficiency and powerful/fast search operations supported. In this paper we analyze, both theoretically and experimentally, the I/O-complexity and the working space of six algorithms for constructing large suffix arrays. Three of them are the state-of-the-art, the other three algorithms are our new proposals. We perform a set of experiments based on three different data sets (English texts, Amino-acid sequences and random texts) and give a precise hierarchy of these algorithms according to their working-space vs. construction-time tradeoff. Given the current trends in model design [12, 33] and disk technology [29, 30], we will pose particular attention to differentiate between "random" and "contiguous" disk accesses, in order to reasonably explain some practical I/O-phenomena which are related to the experimental behavior of these algorithms and that would be otherwise meaningless in the light of other simpler external-memory models.

We also address two other issues. The former concerns with the problem of building word-indexes; we show that our results can be successfully applied to this case too, without any loss in efficiency and without compromising the simplicity of programming so to achieve a uniform, simple and efficient approach to both the two indexing models. The latter issue is related to the intriguing and apparently counterintuitive "contradiction" between the effective practical performance of the well-known BaezaYates-Gonnet-Snider's algorithm [17], verified in our experiments, and its unappealing (i.e., cubic) worst-case behavior. We devise a new external-memory algorithm that follows the basic philosophy underlying that algorithm but in a significantly different manner, thus resulting in a novel approach which combines good worst-case bounds with efficient practical performance.

Keywords: Suffix Array, Large Data Sets, External-memory Model, Text Indexing, Full-text and Word-based Models.

*Part of this work was done while the second author had a Post-Doctoral fellowship at the Max-Planck-Institut für Informatik, Saarbrücken. The work has been supported by EU ESPRIT LTR Project N. 20244 (ALCOM-IT), and by Italian MURST project "Algorithms on Large Data Sets: Science and Engineering"

1 Introduction

In the information age, one of the fastest growing category of databases are the textual databases—like AP-news, Digital Libraries, Genome databases, book collections [15, 6, 35]. Their ultimate impact heavily depends on the ability to *store* and *search* efficiently the information contained into them. The continued decline in the cost of external storage devices (like disks and CD-ROMs) has made nowadays the storage issue not much of a problem, compared to the challenges posed by the fast retrieval of the user-requested informations. In order to achieve this goal, specialized indexing data structures and searching tools have been introduced. Their main idea is to build an *index* that allows to focus the search for a given pattern string only on a very small portion of the text collection. The improvement in the query-performance is paid by the additional space necessary to store the index. Most of the research in this field has been therefore directed to design indexing data structures which offer a good trade-off between the query time and the space usage. The two main approaches are: *word* indexes and *full-text* indexes.

Word-indexes exploit the fact that for natural linguistic texts the universe of distinct words is small. They store all the occurrences of each word in a table that is indexed via a hashing function or a tree structure (they are usually called *inverted lists* [35]). To reduce the size of the table, common words are either not indexed (e.g. the, at, a) or the index is later compressed. The advantage is to support very fast word (or prefix-word) queries, while a weakness is related to the inefficiency of indexing *non* natural-language texts, like DNA-sequences or Chinese texts [13]. (For alternative approaches to word-indexes see [35].)

Full-text indexes have been designed to overcome the limitations above by dealing with arbitrary texts and general queries, at the cost of an increase in the additional space occupied by the underlying indexing data structure. Examples of such indexes are: suffix trees [25, 10], suffix arrays [24] (cfr. PAT-arrays [17]), PAT-trees [17] and String B-trees [14]. They have been successfully applied to fundamental string-matching problems as well text compression [9, 23], analysis of genetic sequences [18] and recently to the indexing of special linguistic texts [13]. General full-text indexes are therefore the natural choice to perform fast complex searches without any restrictions. The most important complexity measures for evaluating their efficiency are [5, 34]: (i) the time and the extra space required to build the index, (ii) the time required to search for a string, and (iii) the space used to store the index. Points (ii) and (iii) have been largely studied in the scientific literature [10, 14, 17, 24, 25]. In this paper, we will investigate Point (i) by addressing the efficient *construction* of full-text indexes on very large text collections. This is a hot topic nowadays [1] because the construction phase may be a bottleneck that can even prevent these indexing tools to be used in large-scale applications. In fact, known construction algorithms are very fast when employed on textual data that fit in the internal memory of computers [4, 24, 31, 22] but their performance immediately degradates when the text size becomes so large that the texts must be arranged on (slow) external storage devices [10, 14].

(Mechanical) Disk vs. (Electronic) Memories. Although disks provide a large amount of space at low cost, their access time is from 10^5 to 10^6 slower than the time to access the internal memory of computers [33]. Unfortunately, the majority of the known algorithms ignore this fact and use disks in the same manner as if all data were fitted into the internal memory and could be accessible in *unit time*. Consequently, they suffer from the so called *I/O bottleneck*: They spend most of the time in moving data to/from the disk. Since our aim is to study the efficiency of algorithms that operate on very large text collections, we need a computational model that captures this *I/O-phenomenon*. Accurate disk models are complex [30, 33], and it is virtually impossible to exploit all the fine points of disk characteristics systematically, either in practice or for algorithmic design. In order to capture in an easy, yet significant, way the differences between the internal (electronic) memory and the external (mechanical) disk, we refer to the external memory model in [33]. Here a computer is abstracted to consist of a *two-level*

[1] Zobel *et al.* [34] say that: "*We have seen many papers in which the index simply 'is', without discussion of how it was created. But for an indexing scheme to be useful it must be possible for the index to be constructed in a reasonable amount of time, ...*".

memory: a fast and small internal memory, of size M, and a slow and arbitrarily large external memory, called *disk*. Data between the internal memory and the disk are transfered in blocks of size B (called *disk pages*). Since disk accesses are the dominating factor in the running time of many algorithms the usual accounting scheme [33] consists of evaluating their asymptotic performance by counting the total number of disk accesses performed during the computation. Although this is a workable approximation for algorithm design, this accounting scheme does not accurately predict the running time of algorithms on real machines because it does not take into account some important specialties of disk systems [29]. In fact, disk access costs have mainly two components: the time to fetch the first bit of requested data (seek time) and the time required to transmit the requested data (transfer rate). Transfer rates are more or less stable but seek times are highly variable [29, 30]. Hence, it is very well known [29, 30, 34] that accessing one page from the disk in most cases decreases the cost of accessing the page succeeding it, so that "bulk" I/Os are less expensive per page than "random" I/Os. This difference becomes much more prominent if we also consider the reading-ahead/buffering/caching optimizations which are common in current disks and operating systems. To deal with these specialties and avoid the introduction of new parameters, we adopt the simple, yet significant, accounting scheme introduced by Farach *et al.* [12]: Let c be a constant, a *Bulk I/O* is the reading/writing of a contiguous sequence of cM/B disk pages; a *random* I/O is any single disk-page access which is not part of a bulk I/O. The performance of the external-memory algorithms is therefore evaluated by measuring: (a) the number of I/Os (bulk and random), (b) the internal running time (CPU time), and (c) the number of disk pages used during the construction process (working space).

Previous Work. For simplicity of exposition, we will use N to denote the size of the whole text collection and we will assume throughout the paper that the index is built on *only one* text, obtained by concatenating all the available texts separated by proper special characters (i.e., endmarkers).

The most famous full-text indexing data structure is the *suffix tree* [25]. In internal memory, a suffix tree can be constructed in $O(N)$ time [25, 12]; in external memory, Farach *et al.* [12] showed that a suffix tree can be optimally constructed within the same I/O-bound as sorting N atomic items; nonetheless, known practical construction algorithms [10] for external memory still operate in a brute-force manner requiring $\Theta(N^2)$ total I/Os in the worst-case. The main limit of these algorithms is inherent in the working space which depends on the text structure, is not predictable in advance and turns out to require between $16N$ and $26N$ bytes (assuming that the $N \leq 2^{32}$ [22, 24]). [2] This makes them impractical even for moderately large text collections (consider what happens for $N \approx 100$ Mbytes, the suffix tree would occupy 1.7Gbytes !). Searching for an arbitrary string of length p takes $O(p)$ time in internal memory (which is optimal for bounded alphabets), but it does not gain any speed up from the block-transfer when the suffix tree is stored on the disk [14].

To circumvent these drawbacks, the *String B-tree* data structure has been introduced in [14]. Searching for an arbitrary pattern string of length p takes $O(p/B + \log_B N)$ random I/Os (which is optimal for unbounded alphabets). The total occupied space is asymptotically optimal, and needs exactly $12.3N$ bytes. The String B-tree is a *dynamic* data structure which supports efficient update operations, but its construction from scratch on a text collection of size N takes $O(N \log_B N)$ random I/Os. Hence, space and construction time may still be a bottleneck in large-scale applications.

Since the space occupancy is a crucial issue when building and using full-text indexes on large text collections, Manber and Myers [24] proposed the *suffix array* data structure (cfr. PAT-array [17]), which consists of an array of pointers to text positions and thus occupies overall $4N$ bytes (thus being 4 times smaller than a suffix tree, and 3 times smaller than a String B-tree). Suffix arrays can be efficiently constructed in $O(N \log_2 N)$ time [24] and $O((N/B)(\log_2 N) \log_{M/B}(N/B))$ random I/Os [1]. In external memory, searching is not as fast as in String B-trees but it still achieves good I/O-performances. Suffix arrays have been recently the subject of experimental investigations in internal memory [24, 31], external memory [17] and distributed memory systems [20, 26]. The motivation has to be probably found in their simplicity, reduced space occupancy, and in the small constants hidden in the big-Oh notation

[2] In [22] the working space has been reduced to $10.1N \div 20N$, assuming also that $N < 2^{27}$.

which make them suitable for achieving reasonable performance on large text collections. Suffix arrays do not require the text to be composed of natural language words, and instead they allow to handle any sequence of symbols. They are also able to perform more complex queries at pattern-matching level including simple words and phrases, regular expressions, and in the presence of errors. Suffix arrays do allow to compute statistics over the indexed texts, like longest repeated substrings and even compare a text against itself to find auto-repetitions with and without errors. Suffix arrays also present some natural advantages over the other indexing data structures for what concerns the construction phase. Indeed, their simple topology (i.e., an array of pointers) avoids at construction time all the problems related to the efficient management of tree-based data structures (like suffix trees and String B-trees) on external storage devices [21]. Additionally and more importantly, efficient practical procedures for building suffix arrays are definitively useful for efficiently constructing suffix trees, String B-trees and other indexing data structures, so that they can allow to overcome their main bottleneck (i.e., expensive construction phase).

Our Contribution. With the exception of some preliminary and partial experimental works [24, 17, 26], to the best of our knowledge, no full-range comparison exists among the known algorithms for building large suffix arrays. This will be the main goal of our paper, where we will theoretically study and experimentally analyze various *six* suffix-array construction algorithms. Some of them are the state-of-the-art in the practical setting [17], others are the most efficient theoretical ones [24, 1], whereas *three* other algorithms are our new proposals obtained either as slight variations of the previous ones or as a careful combination of known techniques which were previously employed only in the theoretical setting. We will study these algorithms by evaluating their working space and their construction complexity both in terms of number of (random and bulk) I/Os and CPU-time. In the design of the new algorithms we will address mainly two issues: (i) simple algorithmic structure, and (ii) reduced working space. The first issue has clearly an impact on the predictability and practical efficiency of the proposed algorithms, which are also *flexible* enough to be used in distributed memory systems. In fact, since our new algorithms will be based on two basic routines—*sorting and scanning of a set of items*—they will immediately provide us with very fast suffix-array construction algorithms also for D-disk arrays systems (thus achieving a speedup factor of approximately D [28]) and clusters of P workstations (thus achieving a speedup factor of approximately P [16]). Additionally, our algorithms will be not faced with the problems of carefully setting some system parameters, as it happens in the results of [20, 26]. The second issue (i.e., space usage) will be also carefully taken into account because the real disk size is limited and thus a large working space could prevent the use of a construction algorithm even for moderately large text collections. Additionally, as Knuth [21][Sect. 6.5] says: "*space optimization is closely related to time optimization in a disk memory*". Our aim will be therefore to keep the working space of our algorithms as small as possible without worsening their total running time.

We will discuss all the algorithms according to these two resources and we will pose particular attention to differentiate between random and bulk I/Os. This will allow us to take into account in the asymptotic analysis the most significant disk characteristics, thus making reasonable predictions on the practical behavior of these algorithms. To validate our conjectures, we will perform an extensive set of experiments based on three text collections—English texts, Amino-acid sequences and random data. The various parameters, accounted for in our theoretical analysis, will allow us to reasonably explain some *interesting I/O-phenomena* which arise during these experiments and which would be otherwise meaningless in the light of other simpler external-memory models (see e.g. [33] for a survey). As a result of the theoretical and experimental analysis, we will give a precise hierarchy of suffix-array construction algorithms according to their working-space vs. construction-time tradeoff; thus providing a wide spectrum of possible approaches for anyone who is interested in building large full-text indexes.

This analysis leads us to additionally address two further issues: construction of word-indexes and worst-case performance of the BaezaYates-Gonnet-Snider's algorithm [17], one of the most effective algorithm in our proposed hierarchy. As far as the former issue is concerned, we show that our new algorithms can be successfully applied to construct word-indexes without any loss in efficiency and without compromising the ease of programming so to achieve a uniform, simple and efficient approach to both the two indexing models. The latter issue deserves much attention and is related to the intriguing, and apparently counterintuitive, "contradiction" between the effective practical performance of the BaezaYates-Gonnet-Snider's algorithm [17], verified in our experiments, and its unappealing (i.e., cubic) worst-case behavior. This fact motivated us to deeply study its algorithmic structure and exploit more fine external-memory models [12] for explaining its experimental performances. This has finally lead us to devise a *new* external-memory construction algorithm that follows the BGS's basic philosophy but in a significantly different manner, thus resulting in a novel approach which combines good practical qualities with efficient worst-case performances.

Map of the Paper. In the next section, we give some basic definitions and fix our notation. In Section 3, we review three well-known algorithms for constructing suffix arrays and analyze their space requirements. The description may appear in some sense much detailed, but our aim is to make this paper as a *self-contained reference* to anyone who is interested in building large suffix arrays. In Section 4, we describe three new external-memory algorithms for constructing suffix arrays on large text collections. In Section 5, we first overview the main features of the external-memory library `LEDA-SM` [11], used to implement all the tested algorithms; and then we introduce our benchmark suite and discuss our experimental settings. In Section 6, we present and discuss the experimental results on the three data sets: Reuters corpus, Amino-acid collection and random texts. In Section 7, we address the problem of constructing word-indexes and show how our results can be easily extended to this indexing model in a natural way. In Section 8, we describe a new algorithm for suffix-array construction which follows the basic philosophy in [17] but in a significantly different manner, thus resulting effective both in theory and in practice. We conclude the paper discussing some open problems and future directions of research.

2 The Suffix Array data structure

Given a string $T[1,N]$, let $T[1,i]$ denote the i-th prefix of T and $T[i,N]$ denote the i-th suffix of T. The symbol $\leq_L$ denotes the *lexicographic order* between any two strings and the symbol $\leq_i$ denotes the lexicographic order between their length-i prefixes: $S \leq_i T$ if and only if $S[1,i] \leq_L T[1,i]$.

The suffix array SA built on the text $T[1,N]$ is an array containing the lexicographically ordered sequence of text suffixes, represented via pointers to their starting positions (i.e., integers). For instance, if $T = ababc$ then $SA = [1,3,2,4,5]$. This way, SA occupies $4N$ bytes if $N \leq 2^{32}$. Manber and Myers [24] introduced this data structure in the early 90s and proposed an interesting algorithm to efficiently search for an arbitrary string $P[1,p]$ in T by taking advantage of the information coded into SA. They proved that all the occ occurrences of P in T can be retrieved in $O(p \log_2 N + occ)$ time in the worst-case using the plain SA, and that this bound can be further improved to $O(p + \log_2 N + occ)$ time if another array of size $4N$ is provided. If SA is stored on the disk, divided into blocks of size B, the search for P takes $O((p/B) \log_2 N + occ/B)$ random I/Os. In practical cases it is $p << B$, so that $p/B = 1$. The simplicity of the search procedure, the small constants hidden in the big-Oh notation, and the reduced space occupancy are the most important characteristics that made this data structure very appealing for practical applications.

3 Constructing a suffix array

We now review three known algorithms for constructing the suffix array data structure on a string $T[1,N]$. We analyze their construction time in terms of CPU-time and number of I/Os (both random and bulk) in the external-memory model. We also address the issues related to their space requirements, by assuming $N \leq 2^{32}$ so that 4 bytes are sufficient to encode a (suffix) pointer. We remark that the working space of all algorithms is *linear* in the length N of the indexed text, and thus *asymptotically optimal.* However, since the constants hidden in the big-Oh notation differ a lot and the available disk space is not unlimited, we

Algorithm	Working space	CPU–time	total number of I/Os
Manber–Myers (Sect. 3.1)	$8N$	$N \log_2 N$	$N \log_2 N$
BaezaYates–Gonnet–Snider (Sect. 3.2)	$8N$	$(N^3 \log_2 M)/M$	$(N^3 \log_2 M)/(MB)$
Doubling (Sect. 3.3)	$24N$	$N(\log_2 N)^2$	$(N/B)\,(\log_{M/B} N/B)\, \log_2 N$
Doubling+Discard (Sect. 4.1)	$24N$	$N(\log_2 N)^2$	$(N/B)\,(\log_{M/B} N/B)\, \log_2 N$
Doubling+Discard+Radix (Sect. 4.1)	$12N$	$N(\log_2 N)^2$	$(N/B)\,(\log_{M/(B \log N)} N)\, \log_2 N$
Construction in L pieces (Sect. 4.3)	$\max\{\frac{24N}{L}, \frac{2NL+8N}{L}\}$	$N(\log_2 N)^2$	$(N/B)\,(\log_{M/B}(N/B))\, \log_2 N$
New BGS (Sect. 8)	$8N$	$N^2(\log_2 M)/M$	(N^2/MB)

Table 1: *The CPU-time and the number of I/Os are expressed in big-Oh notation; the working space is evaluated exactly; L is an integer constant greater than 1. BaezaYates-Gonnet-Snider algorithm (BGS), and its new variant (called* new BGS*), operate via only disk scans, whereas all the other algorithms mainly execute random I/Os. Notice that with a tricky implementation, the working space of BGS can be reduced to $4N$.*

will carefully evaluate the space usage of these algorithms in order to study their practical applicability (see Table 1 for a summary).

3.1 The algorithm of Manber and Myers

It is the fastest theoretically known algorithm for constructing a suffix array in internal memory [24]. It requires $O(N \log_2 N)$ worst-case time and consists of $\lceil \log_2 (N+1) \rceil$ stages, each taking $O(N)$ time. In the first stage, the suffixes are put into buckets according to their first symbol (via radix sort). Before the generic h-th stage starts, the algorithm has inductively identified a sequence of buckets, each containing a set of suffixes. Any two suffixes in the same bucket share the first 2^{h-1} characters, whereas any two suffixes in two different buckets are $\leq_L$-sorted according to the bucket-ordering (initially, we have just one bucket containing all of T's suffixes). In stage h, the algorithm sorts lexicographically the suffixes of each bucket according to their first 2^h characters, thus forming new smaller buckets which preserve the inductive hypothesis. After the last stage, all the buckets will contain one suffix, thus giving the final suffix array. The efficiency of this algorithm depends on how fast is the sorting step in a generic stage. Manber and Myers [24] showed how to perform it in linear time by using only two integer arrays, for a total of $8N$ bytes. If this algorithm is used in a virtual memory setting, it tends to perform $\Theta(N \log_2 N)$ random I/Os.

3.2 The algorithm of BaezaYates-Gonnet-Snider

The algorithm [17] computes incrementally the suffix array SA of the text string $T[1, N]$ in $\Theta(N/M)$ stages. Let $\ell < 1$ be a positive constant fixed below, and assume to set $m = \ell M$. The latter parameter will denote the size of the text pieces loaded in memory at each stage. We also assume for the sake of presentation that m divides N.

The algorithm maintains at each stage the following invariant: *At the beginning of stage h, where $h = 1, 2, \ldots, N/m$, the algorithm has stored on the disk an array SA_{ext} containing the sequence of the first $(h-1)m$ text suffixes ordered lexicographically and represented via their starting positions in T.*

During the hth stage, the algorithm *incrementally updates SA_{ext}* by properly inserting into it the text suffixes which start in the substring $T[(h-1)m+1, hm]$, and by maintaining their lexicographic order. This preserves the invariant above. Hence, after all the N/m stages are executed, it is $SA_{ext} = SA$. We are therefore left with showing how the generic h-th stage works.

The text substring $T[(h-1)m+1, hm]$ is loaded into internal memory, and the suffix array SA_{int} containing only the suffixes starting in that text substring is built by possibly accessing the disk, if needed. [3] Then, SA_{int} is merged with the current SA_{ext} to produce the new array and preserve the

[3] The comparison between any two suffixes can require to access the substring $T[hm+1, N]$, which is still on disk, thus inducing some random I/Os.

invariant. This merging process is executed in two steps with the help of a counter array $C[1, m+1]$. In the former step, the text T is scanned rightwards and the lexicographic position j of each text suffix $T[i, N]$, with $1 \leq i \leq (h-1)m$, is determined in SA_{int} via a binary search. The entry $C[j]$ is then incremented by one unit in order to record the fact that $T[i, N]$ lexicographically lies between the $SA_{int}[j-1]$-th and the $SA_{int}[j]$-th suffix of T. In the latter step, the information kept in the array C is employed to quickly merge SA_{int} with SA_{ext}: entry $C[j]$ indicates how many consecutive suffixes in SA_{ext} follow the $SA_{int}[j-1]$-th text suffix and precede the $SA_{int}[j]$-th text suffix. This implies that a simple disk scan of SA_{ext} is sufficient to perform such a merging process. At the end of these two steps, the invariant on SA_{ext} has been properly preserved so that h can be incremented and the next stage can correctly start.

Some comments are in order at this point. It is clear that the algorithm proceeds by mainly executing two disk scans: one is performed to load $T[(h-1)m+1, hm]$ in internal memory, and another is performed to merge SA_{int} and SA_{ext} via the counter-array C. However, some random disk accesses may be necessary in two distinct situations: either when SA_{int} is built or when the lexicographic positions of each text suffix $T[i, N]$ is determined in SA_{int}. In both these two cases, we may need to compare a pair of text suffixes which share a long prefix not entirely available in internal memory (i.e., out of $T[(h-1)m+1, hm]$). In the worst case, this comparison will require two sequential disk scans (initiated at the starting positions of these two suffixes) taking $O(N/M)$ bulk I/Os.

As far as the worst-case I/O-complexity is concerned, let us consider the pathological case in which we have $T = a^N$. Here, we need $O((m \log_2 m)N/m)$ bulk I/Os to build SA_{int}; $O((h-1)m(\log_2 m)(N/m))$ bulk I/Os to compute the array C; and $O(hm/M) = O(h)$ bulk I/Os to merge SA_{int} with SA_{ext}. No random I/Os are executed, so that the global number of bulk I/Os is $O((N^3 \log_2 M)/M^2)$. Since the algorithm processes each loaded block, we may assume that it takes $\Theta(M)$ CPU-time to operate on each of them thus requiring $O((N^3 \log_2 M)/M)$ overall CPU-time. The total space required is $4N$ bytes for SA_{ext} and $8m$ bytes for both C and SA_{int}; plus m bytes to keep $T[(h-1)m+1, hm]$ in internal memory (ℓ's value is derived consequently). The merging step can be easily implemented using some extra space (indeed additional $4N$ bytes), or by employing just the space allocated for SA_{int} and SA_{ext} via a more tricky implementation. We adopt for simplicity the former strategy.

Since the worst-case number of total I/Os is cubic, a purely theoretical analysis would classify this algorithm much less interesting than the others discussed in the following sections (see Table 1). But there are some considerations that are crucial to shed new light on it, and look at this algorithm from a different perspective. First of all, we must observe that in practical situations, it is very reasonable to assume that each suffix comparison finds in internal memory all the (usually, constant number of) characters needed to compare the two involved suffixes. Consequently, the practical behavior is more reasonably described by the formula: $O(N^2/M^2)$ bulk I/Os and $O((N^2 \log_2 M)/M)$ CPU time. Additionally, the accounting scheme adopted in this paper evidences some positive features that would undergo unobserved without this accounting model. Indeed, the analysis above has pointed out that all the I/Os are *sequential* and that the actual number of random seeks is actually $O(N/M)$ (i.e., at most a constant number per stage). Consequently, the algorithm takes fully advantage of the large bandwidth of current disks and of the high computation-power of current processors [29, 30]. Moreover, the reduced working space facilitates the prefetching and caching policies of the underlying operating system (remember Knuth's quote cited in the Introduction) and finally, a careful look to the algebraic calculations shows that the constants hidden in the big-Oh notation are very small. As a result, the adopted accounting scheme does not label the BGS-algorithm as "worse" but drives us to conjecture good practical performances, leaving the final judgment to depend on disk and system specialties. These aspects will be addressed in Section 6.

Implementation Issues. The implementation of this algorithm is tricky if one wants to take into account various pathological cases occurring in the merging step of SA_{int} and SA_{ext}, and in the construction of SA_{int}. These cases could lead the algorithm to execute many I/Os that would *apparently* be classified as random (according to the accounting scheme) but which are likely to be buffered by the system and thus can be executed much faster. Our main idea is twofold:

- The array SA_{int} is built only on the first $m-B$ suffixes of $T[(h-1)m+1, hm]$, thus we discard the last B suffixes (one page) of that text piece. These discarded suffixes will be (re)considered in the next stage. In such a way, during the construction of SA_{int}, each suffix is guaranteed to have a *prefix of at least* B characters available in internal memory, hence reducing *significantly* the probability of a page fault for a suffix comparison.

- To merge SA_{int} and SA_{ext} we need to load the suffixes starting in $T[1,(h-1)m]$ and search them into SA_{int}. These suffixes are loaded via a rightward scan of T, which brings into internal memory text pieces whose size is *twice* the size of a bulk I/O. Then, only the suffixes starting in the *first half* of this piece are searched in SA_{int}, thus guaranteeing to have in internal memory at least cM characters for their comparison.

These two simple tricks avoid some *"border cases"* which are very likely to induce random I/Os and that instead can be easily canceled via a careful programming. We experimented a dramatic reduction in the total number of random I/Os and a consequent significant speedup in the final performance of the implemented BGS-algorithm (see Section 6).

3.3 The doubling algorithm

This algorithm was introduced in [1] as a variant of the *labeling technique* of [19], properly adapted to work in external memory. The main idea is first to *logically* pad T with N *special* blank characters which are smaller than any other text character; and then to assign names (i.e. small integers) to the power-of-two length substrings of $T[1,2N]$ in order to satisfy the so called *lexicographic naming property*: Given two substrings α and β of length 2^h occurring in T, it is $\alpha \leq_L \beta$ if and only if the name of α is smaller than the name of β. These names are computed inductively by exploiting the following observation: the lexicographic order between any two substrings α, β of length 2^h can be obtained by splitting them into two equal-length parts $\alpha = \alpha_1\alpha_2$ and $\beta = \beta_1\beta_2$, and by using the order between the two pairs of names inductively assigned to (α_1, α_2) and (β_1, β_2). After $q = \lceil \log_2 (N+1) \rceil$ stages, the algorithm has computed the lexicographic names for the 2^q-length substrings of $T[1,2N]$ starting at positions $1,\ldots,N$ (where $2^q \geq N$). Consequently, the order between any two text suffixes, say $T[i,N]$ and $T[j,N]$, can be obtained in constant time by comparing the lexicographic names of $T[i,i+2^q-1]$ and $T[j,j+2^q-1]$. This is the rationale behind the doubling algorithm in [1], whose implementation details are sketched below.

At the beginning, the algorithm scans the string $T[1,N]$ and creates a list of N tuples each consisting of four components, say $\langle 0,0,i,T[i]\rangle$ (the third component is fixed throughout the algorithm). [4] During all $q = \lceil \log_2 (N+1) \rceil$ stages, the algorithm manipulates these tuples by preserving the following invariant: *At the beginning of stage h (initially $h=1$), tuple $\langle *,*,j,n_j\rangle$ keeps some information about the substring $T[j,j+2^{h-1}-1]$ and indeed n_j is its lexicographic name.* [5] After the last q-th stage, the suffix array of T is obtained by executing two steps: (i) sort the tuples in output from stage q according to their fourth component; (ii) construct SA by reading from left-to-right the third component of the tuples in the ordered sequence.

We are therefore left with showing how stage h can preserve the invariant above by computing the lexicographic names of the substrings of length 2^h with the help of the names inductively assigned to the 2^{h-1}-length substrings. This is done in four steps as follows.

1. The N tuples (in input to stage h) are sorted according to their third component, thus producing a list such that its i-th tuple has the form $\langle *,*,i,n_i\rangle$ and thus keeps the information regarding the substring $T[i,i+2^{h-1}-1]$.

[4] The blank characters are only *logically* appended to the end of T.

[5] In what follows, the symbol $*$ is used to denote an arbitrary value for a component of a tuple, which is actually not important in the discussion.

2. This list is scanned rightwards and the i-th tuple is substituted with $\langle n_i, n_{i+2^{h-1}}, i, 0\rangle$, where $n_{i+2^{h-1}}$ is the value contained in the fourth component of the $(i+2^{h-1})$-th tuple in the list. [6]
3. The list of tuples is sorted according to their first two components (lexicographic naming property).
4. The sorted sequence is scanned rightwards and different tuples are assigned (in their fourth component) with *increasing* integer numbers. This way, the lexicographic naming property is preserved for the substrings of T having length 2^h.

The correctness of this approach immediately derives from the lexicographic naming property and from the invariant preserved at every stage h which actually guarantees that:

Property 1 *Each tuple $\langle *, *, i, *\rangle$ contains some compact* lexicographic information *about the 2^h-length prefix of the suffix $T[i, N]$.*

As far as the I/O complexity is concerned, each stage applies twice a sorting routine (steps 1 and 3) and twice a scanning routine (steps 2 and 4) on a sequence of N tuples. The total number of random I/Os is therefore $O(sort(N)\ \log_2 N)$, and the total number of bulk I/Os is $O((N/M)\ \log_2 N)$, where $sort(N) = (N/B) \log_{M/B}(N/B)$ is the (random) I/O-complexity of an optimal external-memory sorting algorithm [33]. The CPU-time is $O(N \log_2^2 N)$ since we perform $O(\log_2 N)$ sorting steps on N items. As far as the space complexity is concerned, this algorithm sorts tuples of four components, each consisting of an integer (i.e., 4 bytes). Hence, it seems that 16 bytes per tuple are necessary. Instead, by carefully redesigning the code it is possible to save one entry per tuple, thus using only 12 bytes[7]. In summary, the total space complexity is $24N$ bytes because the implementation of the multiway mergesort routine [21], used in our experiments to sort the tuples, needs $2Xb$ bytes for sorting X items of b bytes each (see Section 5).

Two practical improvements are still possible. The first improvement can be obtained by coding four consecutive characters of T into one integer before that the first stage is executed. This allows the saving of the first two stages and hence overall four sorting and four scanning steps. This improvement is not negligible in practice due to the time required by the sorting routine (see Figure 1). The second improvement comes from the observation that it is not necessary to perform $\Theta(\log_2 N)$ iterations, but the doubling process can be stopped as soon as all the 2^h-length substrings of T are different (i.e. all tuples get different names in step 4). This modification does not change the worst-case complexity of the algorithm, but it ensures that only six stages are usually sufficient for natural texts [10].

4 Our three new proposals

In this section we introduce three new algorithms which asymptotically improve the previously known ones by offering better trade-offs between total number of I/Os and working space. Their algorithmic structure is simple because it is based only upon *sorting* and *scanning* routines. This feature has two immediate advantages: The algorithms are expected to be fast in practice because they can benefit from the prefetching of the disk [29, 30]; they can be easily adapted to work efficiently on D-disk arrays and clusters of P workstations. It suffices to plug-in proper sorting/scanning routines to obtain a speed-up of a factor D [28] or P [16], approximately (cfr. [20, 26]).

[6] The rationale behind this step is to represent each substring $T[i, i+2^h-1]$ by means of the lexicographic names assigned to its prefix and its suffix of length 2^{h-1}. These names are inductively available at the beginning of stage h.

[7] One can drop the fourth entry of the tuple as it can be temporarily coded in the first and second entries.

4.1 Doubling combined with a discarding stage

Our first new algorithm is based on the following observation: *In each stage of the doubling approach, all tuples are considered although the final position in SA of some of them might be already known.* Therefore all those tuples could be discarded from the succeeding sorting steps, thus reducing the overall number of operations, and hence I/Os, executed in the next stages (cfr. [31]). Although this discarding strategy does not determines an asymptotic speed up on the overall performance of the algorithm, it is nonetheless expected to induce a significant improvement on experimental data sets because it tends to reduce the number of items on which the sorting/scanning routines are required to work on. [8]

The main idea is therefore to identify in the step 4 of the doubling algorithm (see Section 3.3), all the tuples whose final lexicographic position can be inferred using the available information, and then discard them from the next sorting stages. However, these tuples cannot be completely excluded because they might be necessary in the step 1 of the succeeding stages in order to compute the names of longer prefixes of suffixes whose position has been not yet established. In what follows, we exploit Property 1 to cope with this problem.

As in the original doubling algorithm, we assume that a tuple has three entries (the fourth one has been dropped, see Section 3.3), and we call a tuple *finished* if its second component is set to -1. The new algorithm inductively keeps two lists of tuples: `FT` (finished tuples) and `UT` (unfinished tuples). The former is a sorted list of tuples corresponding to suffixes whose final position in SA is known; they have the form $\langle pos, -1, i\rangle$, where pos is the final position of suffix $T[i, N]$ in SA (i.e., $SA[pos] = i$). `UT` is a list of tuples $\langle x, y, i\rangle$, corresponding to suffixes whose final position is not yet known, and are such that $x, y \geq 0$ denote *lexicographic names* and $T[i, N]$ is the suffix to which this (unfinished) tuple refers.

At the beginning, the algorithm creates the list `UT` with tuples having the form $\langle 0, T[i], i\rangle$, for $1 \leq i \leq N$, sets `FT` to the empty list and initializes the counter $j = 0$. Then, the algorithm proceeds into stages each consisting of the following steps:

1. **Sort** the tuples in `UT` according to their first two components. If `UT` is empty goto step 6.

2. **Scan** `UT`, mark the "finished" tuples and assign new names to all tuples in `UT`. Formally, a tuple is "finished" if it is preceded and followed in `UT` by two tuples which are different in at least one of their first two components; in this case, the algorithm marks "finished" the current tuple by setting its second component to -1. The new names for all tuples of `UT` are computed differently from what was done in step 4 of the doubling algorithm (see Section 3.3). Indeed, the first component of a tuple $t = \langle x, y, *\rangle$ is now set equal to $(x + c)$, where c is the number of tuples that precede t in `UT` and have the form $\langle x, p, *\rangle$ with $p \neq y$.

3. **Sort** `UT` according to the third component of its tuples (i.e., according to the starting position of the corresponding suffix).

4. **Merge** the lists `UT` and `FT` according to the third component of their tuples. `UT` will keep the final merged sequence, whereas `FT` will be emptied.

5. **Scan** `UT` and for each *not-finished* tuple $t = \langle x, y, i\rangle$ (with $y \neq -1$), take the next tuple at distance 2^j (say $\langle z, *, i + 2^j\rangle$) and change t to $\langle x, z, i\rangle$. If a tuple is marked "finished" (i.e., $y = -1$), then it is discarded from `UT` and put into `FT`. Finally set $j = j + 1$ and go to step 1.

6. (`UT` is empty) `FT` is sorted according to the first component of its tuples. The third component of the sorted tuples read rightwards, give the final suffix array.

The correctness can be proved by the following invariant:

Property 2 *At a generic stage j ($j \geq 0$), the execution of step 2 ensures that a tuple $t = \langle x, y, i\rangle$ satisfies the property that x is the number of T's suffixes whose prefix of length 2^j is strictly smaller than $T[i, i + 2^j - 1]$.*

[8] Knuth [21][Sect. 6.5] says: "*space optimization is closely related to time optimization in a disk memory*".

Proof: Before step 2 is executed, x inductively accounts for the number of suffixes in T whose 2^{j-1}-length prefix is lexicographically smaller than the corresponding one of $T[i,N]$. At the first stage ($j=0$), the algorithm has indeed safely set the first component of each tuple to 0. When step 2 is executed and tuple t is processed, the variable c accounts for the number of suffixes whose 2^{j-1}-length prefix is equal to $T[i,i+2^{j-1}-1]$ but their 2^j-length prefix is smaller than $T[i,i+2^j-1]$. From the inductive hypothesis on the value of x, it then follows that the new value $(x+c)$ correctly accounts for the number of suffixes of T whose 2^j-length prefix is lexicographically smaller than the corresponding one of $T[i,N]$. ∎

The logic underlying the algorithm above is similar to the one behind the original doubling algorithm (see Section 3.3). However, here the new names are assigned in step 2 by following a completely different approach which guarantees not only the lexicographic naming property but also a proper coding of some useful informations (Property 2). This way when a tuple is marked "finished", its first component correctly gives the final position in SA of the corresponding suffix (denoted by its third component). Therefore, the tuple can be safely discarded from `UT` and put into `FT` (step 5).

Doubling combined with the discarding strategy performs $O((N/B)(\log_{M/B}(N/B))\log(N))$ random I/Os, $O((N/M)\log_2 N)$ bulk I/Os, and occupies $24N$ bytes (see Section 3.3), exactly the same I/O-complexity as the Doubling algorithm. In our implementation, we will also use the compression scheme discussed at the end of Section 3.3, to save the first two stages and thus four sorting and four scanning steps. As conjectured at the beginning of this section, we expect that the discarding strategy induces a significant speed-up in the practical performance of the Doubling algorithm.

4.2 Doubling+Discard and Radix Heaps

Although the doubling technique gives the two most I/O-efficient algorithms for constructing large suffix arrays, it has the major drawback that its working space is large (i.e. $24N$ bytes) compared to the other known approaches (see also Table 1). This is due to the fact that it uses an external mergesort [21] for ordering the list of tuples and this requires an auxiliary array to store the intermediate results. Our new idea here is to reduce the overall space requirements by making use of an external version of the radix heap data structure introduced in [2]. This data structure [8] uses an exact number N/B of disk pages to store N items, and it is a *monotone integer priority queue* that supports the insertion of a new item in amortized $O(1/B)$ I/Os and the deletion of the item having minimum priority in amortized $O(1/B\log_{M/(B\log_2 V)} V)$ I/Os, where V is the maximum priority value. Hence radix heaps are space efficient but their I/O-performance degenerates when V is large. Our new construction algorithm replaces the mergesort in the Doubling+Discard algorithm (see steps 1 and 3 in Section 4.1) with a sorting routine based on that external radix heap. This reduces the overall required space to $12N$ bytes, but at the cost of increasing the I/O–complexity to $O((N/B)(\log_{M/(B\log_2 N)} N))\log_2 N)$ random I/Os (and $O((N/M)\log_2 N)$ bulk I/Os), because $V=N$ in the step 3 of Section 4.1 (indeed, the third component of the sorted tuples ranges in $[1,N]$). We observe that this algorithm should outperform the doubling approach during the first stages because the range of assigned names, and thus the value of V, is sufficiently small to take advantage from the radix-heap structure. On the other hand, the algorithm performance degenerates as more stages are executed because V becomes larger and larger. It is therefore interesting to experimentally investigate this solution since it significantly saves space and is expected to behave well in practice even in the light of the reduction in the number of tuples to be sorted in each stage. Probably, this reduction compensates the time increase of the radix heap approach and thus it is worth to be experimented on real data (see Section 6).

4.3 Construction in L pieces

The approaches described before are I/O-efficient but they use at least $8N$ bytes of working space. If the space issue is a primary concern, and we still wish to keep the total number of I/Os small, different approaches must be devised that require much less space but still guarantee good I/O-performances [9]. In this section, we describe one such approach which improves over all previous algorithms in terms of both I/O complexity, CPU time and space occupancy. It constructs the suffix array into *pieces of equal sizes* and thus it turns out useful either when the underlying application *does not* need the suffix array as a unique data structure, but allows to keep it in a distributed fashion [7]; or when we operate in a distributed-memory environment [20, 26].

The most obvious way to achieve this goal might be to partition the original text string $T[1,N]$ into equal-length substrings and then apply on these pieces any known suffix-array construction algorithm. However, this approach should cope with the problem of correctly handling the suffixes which *start close to the border* of any two text pieces. To circumvent this problem, some authors [20, 26] reduce the suffix array construction process to a string sorting process associating to each suffix of T, its prefix of length X, and then sorting N strings of length X each. Clearly, the correctness of this approach heavily depends on the value of X which also influences the space occupancy (it is actually NX bytes). Statistical considerations and structural informations about the underlying text might help, but anyway the choice of the parameter X strongly influences the final performance (see e.g. [20, 26]).

The approach we introduce below is very simple and applies in a different way, useful for practical purposes, a basic idea known so far only in the theoretical setting (see e.g. [12]). Let us denote by $\mathcal{A}_{sa}$ any external-memory algorithm for building a suffix array, $\mathcal{A}_{string}$ any external-memory algorithm for sorting a set of strings, and let L be a constant integer parameter to be fixed later. For simplicity of exposition, we assume that N is a multiple of L, and that T is *logically* padded with L blank characters.

The new approach constructs L suffix arrays, say $SA_1, SA_2, \ldots, SA_L$ each of size $\Theta(N/L)$. Array SA_i stores the lexicographically ordered sequence of suffixes $\{T[i,N], T[i+L,N], T[i+2L,N], \ldots,\}$. The logic underlying our algorithm is to first construct SA_L, by using $\mathcal{A}_{sa}$ and $\mathcal{A}_{string}$, and then derive all the others arrays $SA_{L-1}, SA_{L-2}, \ldots, SA_1$ by means of a simple external-memory algorithm for sorting *triple of integers* (e.g., multiway mergesort [21]).

The suffix array SA_L is built in two main stages: In the first stage, the string set $S = \{T[L, 2L-1], T[2L, 3L-1] \ldots, T[N-L, N-1], T[N, N+L-1]\}$ is formed and lexicographically sorted by means of algorithm $\mathcal{A}_{string}$. In the second stage, a compressed text T' of length N/L is derived from $T[L, N+L-1]$ by replacing each string having the form $T[iL, (i+1)L-1]$, for $i \geq 1$, with its *rank* in the *sorted* S. Then, algorithm $\mathcal{A}_{sa}$ builds the suffix array SA' of T', and finally derives SA_L by setting $SA_L[j] = SA'[j] \times L$, for $j = 1, 2, \ldots, N/L$.

Subsequently, the other $L-1$ suffix arrays are constructed by exploiting the following observation: *Any suffix $T[i+kL, N]$ in SA_i can be seen as the concatenation of the character $T[i+kL]$ and the suffix $T[i+1+kL, N]$ occurring into SA_{i+1}.* So that, if SA_{i+1} is known, the order between $T[i+kL, N]$ and $T[i+hL, N]$ can be obtained by comparing the two pairs of integers $\langle T[i+kL], pos_{i+1+kL}\rangle$ and $\langle T[i+hL], pos_{i+1+hL}\rangle$, where pos_s is the position in SA_{i+1} of suffix $T[s,N]$. This immediately means that the construction of SA_i can be reduced to sorting $\Theta(N/L)$ tuples, once SA_{i+1} is known.

Sorting via $\mathcal{A}_{string}$ the short strings of length L takes $O(Sort(N))$ random I/Os and $2N + 8N/L$ bytes of storage, where $Sort(N) = (N/B)\ \log_{M/B}(N/B)$ [1]. Building the L suffix arrays SA_i takes $O(Sort(N/L)\ \log_2(N/L) + L\, Sort(N/L)) = O(Sort(N)\ \log_2 N)$ random I/Os, $O(N/M \log_2(N/M))$ bulk I/Os and $24N/L$ bytes. Of course, the larger is the constant L, the larger is the number of suffix arrays that will be constructed, but the smaller is the working space required. By setting $L = 4$, we get an interesting algorithm for constructing large suffix arrays: it needs $6N$ working space, $O(Sort(N)\ \log_2 N)$ random I/Os and $O(N/M \log_2(N/M))$ bulk I/Os. Its practical performance will be evaluated in Section 6. Notice that this approach builds four suffix arrays, thus its query performance is slowed down by a constant factor four, but this is practically negligible in the light of suffix-array search performance.

[9] See the footnote 8 and refer to [34] where Zobel *et al.* say that: "*A space-economical index is not cheap if large amounts of working storage are required to create it.*"

5 Our experimental settings

5.1 An external-memory library

We implemented all algorithms discussed so far by using a recently developed external-memory library of algorithms and data structures called LEDA-SM [11] (an acronym for "LEDA for Secondary Memory"). This library is an extension of the internal-memory library *LEDA* [27] and therefore follows its main underlying ideas. One of them is *portability* to a variety of platforms and another one is *high level specification* of data structures. Library LEDA-SM consists of a sizeable collection of efficient data structures and algorithms explicitly designed to work in an external memory setting. The system underlying LEDA-SM reflects a real view of the external memory model: The internal memory is directly provided by the internal memory of the computer; whereas the D abstract disks are modeled by the file system, which also provides proper tools for implementing the low-level I/O via block transfers. Each disk is modeled with a *single file* and it is divided into logical blocks of a fixed size B (disk pages). The size of this file is fixed, thus modeling the fact that real disk space is bounded. Since LEDA-SM uses the file system, it explicitly takes advantage of the underlying *I/O-buffering* and *read-ahead* strategy at no additional implementation effort.[10] The part of the system which controls the I/O is the so called *external memory manager*. It is implemented as an interface to which are connected all high level data structures and algorithms, thus guaranteeing that the I/O and the management of the data on the disks is hidden to the programmer. Nevertheless, the programmer can still keep track of the number of disk accesses (bulk, random and total) performed by his/her algorithms, because the system allows the explicit counting of the number of both writes and reads to the D disks. At the moment, LEDA-SM provides the possibility to choose among five different file system access methods, namely standard I/O (stdio), system call I/O (syscall), asynchronous I/O (aio), serial file I/O (sfio) and memory mapped I/O (mmap_io) (see [32] for a deeper discussion). Both the external memory manager and the data structures provided by LEDA-SM are implemented in C++ as a set of template classes and functions. The specialty of LEDA-SM's data structures is that during their constructions it is possible to specify (and therefore control) the maximum amount of internal memory that they are allowed to use. Combining this feature with the counting of I/Os, library LEDA-SM allows the programmer to experimentally investigate how the model parameters M and B influence the performance of an external-memory algorithm.

For what concerns the implementation of our suffix-array construction algorithms, we used the external array data structure and the external sorting/scanning algorithms provided by LEDA-SM library. In particular, we used an implementation of multiway mergesort that needs $2Xb$ bytes for sorting X items of b bytes each (see Figure 1). The other in-core algorithms and data structures used in our experiments are taken from the LEDA-library. To avoid that the internal-memory size prevents the use of Manber-Myers' algorithm on large text collections, we run it in a virtual memory setting by using swap space. All other algorithms are not faced with this problem because they are directly designed to work in external memory. All the algorithms discussed in this paper are available and necessitate both the LEDA and LEDA-SM libraries (please contact one of the authors).

5.2 System parameters

The computer used in our experiments is a SUN ULTRA-SPARC 1/143 with 64 Mbytes of internal memory running the SUN Solaris 2.5.1 operating system. It is connected to one single Seagate Elite-9 SCSI disk via a fast-wide differential SCSI controller. The external memory library LEDA-SM provides logical disks of maximum size of 2 Gbytes (this is the file size limit of Solaris 2.5.1), the disks are divided into blocks of $B = 8$ Kbytes. We have chosen the standard I/O (stdio) for the file system accesses because it is available on almost every system. We have designed our external-memory algorithms so that they use approximately 36 Mbytes of internal memory. This is obtained by properly choosing the internal memory size dedicated to the external memory data structures (e.g. external arrays and mergesort

[10] As the file system is allowed to buffer disk pages, some of the disk requests can immediately be satisfied by the I/O–buffer. This notably speeds up the real performances of the external-memory algorithms, as we discussed in Section 1.

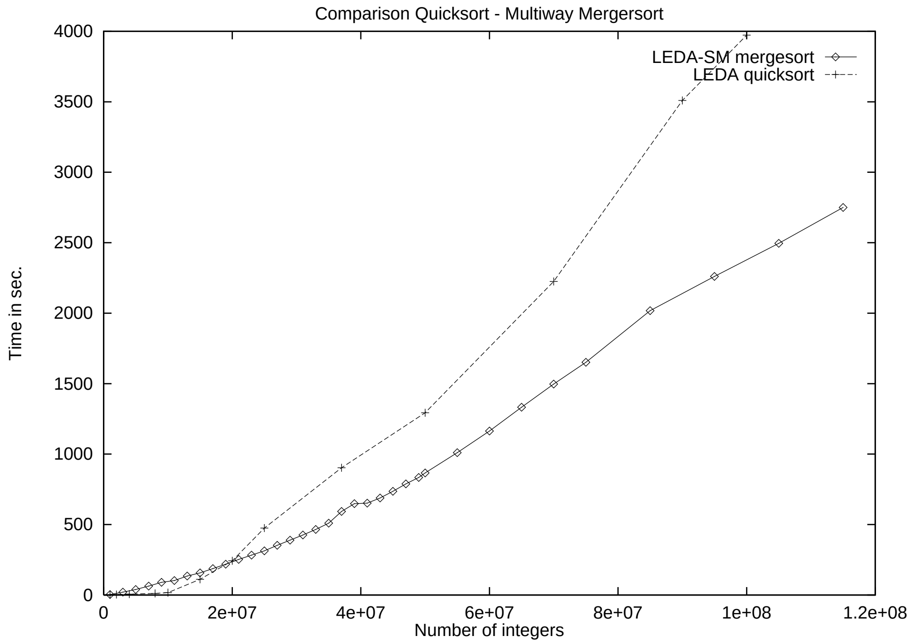

Figure 1: *Comparison between LEDA-quicksort and LEDA-SM multiway mergesort. Quicksort is allowed to use at most 64 Mbytes of internal memory, the further needs of memory are satisfied by using swap space. Multiway mergesort is only allowed to use 12 Mbytes of internal memory.*

buckets). Thus, we guarantee that the studied algorithms do not incur in the *paging phenomenon* even for accessing their internal data structures and furthermore, that there is left room in internal memory to keep the I/O buffers of the operating system.

5.3 Choosing the bulk-load size

Modern SCSI disk drives are equipped with a large cache, which is used to hide the speed gap between the SCSI bus and the disk drive. The disk drive *prefetches* some data into the cache before it initiates the transfer of the requested data from cache to main memory, in order to hide the difference between the rotational delay of the disk and the bandwidth of the bus. Moreover, during read accesses the disk prefetches further data into the cache in order to hopefully serve the next read requests directly from the disk cache, thus avoiding the costly (mechanical) *seek operation*.

According to the accounting scheme we adopted in this paper, we need to choose a reasonable value for the *bulk load size* (notice that the parameters B and M are fixed by default according to the computer features). Since the transfer rates are more or less stable (and currently large) while seek times are highly variable and costly (because of their mechanical nature), our idea is to choose a value for the bulk size which allows to hide the *extra cost* induced by the seek step when data are fetched. This would allow us to access "uniformly" to every datum stored on the disk, thus working at the highest speed allowed by its bandwidth. In some sense, we would like to *hide* the mechanical nature of the disk system [29].

Let the transfer time be described by the formula `t_seek + bulk_size/disk_bandwidth` [30]. We wish that the first term is much smaller than the second one. Consequently, from one side we should increase `bulk_size` as much as possible (at maximum M/B), but from the other side a large `bulk_size` might reduce the significance of our accounting scheme because many *long* sequential disk scans could result shorter than `bulk_size` and thus counted as 'random', whereas they nicely exploit the disk caching and prefetching strategies [12]. Hence a proper "tuning" of this parameter is needed according to the mechanical features of the underlying disk system.

In the disk used for our experiments, the average `t_seek` is 11 msecs, the `disk_bandwidth` is

7 Mbytes/sec. We have therefore chosen `bulk_size` = 64 disk pages, for a total of 512 Kbytes. It follows that `t_seek` is 15% of the total transfer time needed for a bulk I/O. Additionally, the bulk size of 512 Kbytes allows us to achieve 81% of the maximum data transfer rate of our disk while keeping the service time of the requests still low.

According to our considerations above (see also [29]), we think that this is a reasonable choice even in the light of the disk cache size of 1 Mbytes. Surprisingly, we also noticed in our experiments that this value allows us to catch the execution of numerous bulk I/Os by subroutines (e.g. multiway mergesort) which were defined "mainly random" at a theoretical investigation. Clearly, other values for `bulk_size` might be chosen and experimented, thus achieving different trade-offs between random/bulk disk accesses. However, the *qualitative* considerations on the algorithmic performance drawn at the end of the next section will remain mostly unchanged, thus fitting our experimental desires.

5.4 Textual data collections

For our experiments we collected over various WEB sites three textual data sets. They consist of:

- The Reuters corpus[11] together with other natural English texts whose size sum up to 26 Mbytes. This collection has the nice feature of presenting long repeated substrings.
- A set of amino-acid sequences taken from a SWISSPROT database[12] summing up to around 26 Mbytes. This collection has the nice feature of being an unstructured text so that full-text indexing is the obvious choice to process these data.
- A set of *randomly generated* texts consisting of three collections: one formed by texts randomly drawn from an alphabet of size 4, another formed by texts randomly drawn from an alphabet of size 16, and the last one formed by texts randomly drawn from an alphabet of size 64. These collections have two nice features: they are formed by unstructured texts, and they constitute a good test-bed to investigate the influence of the *length of the repeated substrings* on the performance of some studied algorithms. For each alphabet size we consider texts of 25 Mbytes and 50 Mbytes, thus further enlarging the spectrum of text sizes on which the algorithms are tested.

An attentive reader might observe that, although this paper is on constructing suffix arrays on *large* text collections, our experimental data sets seem indeed *not very large* ! If we just look to their sizes, the involved numbers are actually not big (at most 50 Mbytes); but, as it will soon appear clear, our data sets are sufficiently large to evaluate/compare in a fair way the I/O-performance of the analyzed algorithms, and investigate their *scalability* in an external-memory setting. In fact, the suffix array SA needs $4N$ bytes to index a text of length N. Hence, the text plus SA globally occupy 200 Mbytes, when $N = 50$ Mbytes. Additionally, each of the algorithms discussed in our paper requires at least $8N$ bytes of working space; this means 400 Mbytes for the largest text size. In summary, *more than* 600 Mbytes will be used during the overall construction of SA in each of the experimented algorithms, when $N = 50$ Mbytes ! Now, since 64 Mbytes is the size of the available internal memory of our computer, all the experiments will run on disk, and therefore the performance of the studied algorithms will properly reflect their I/O-behavior.

6 Experimental Results

We now comment the results obtained for our six different construction algorithms: Manber and Myers' algorithm (MM), BaezaYates-Gonnet-Snider's algorithm (BGS), original doubling (Doubl), doubling with discarding (Doubl+Disc), doubling with discarding and external radix heaps (Doubl+Disc+Radix), and the 'construction into pieces' approach (L-pieces). The overall results are reported in Figure 2, and they are detailed in Tables 2 and 3 below.

[11]We used the text collection "Reuters-21578, Distribution 1.0" available from David D. Lewis' professional home page, currently: `http://www.research.att.com/~lewis`

[12]See the site: `http://www.bic.nus.edu.sg/swprot.html`

The Reuters corpus						
N	MM	BGS	Doubl	Doubl+Disc	Doubl+Disc+Radix	L-pieces
1324350	67	125	828	982	1965	331
2578790	141	346	1597	1894	3739	582
5199134	293	1058	3665	4070	7833	1119
10509432	223200	4808	8456	8812	16257	2701
20680547	–	16670	23171	20434	37412	5937
26419271	–	27178	42192	28937	50324	7729
The Amino-Acid Test						
26358530	–	20703	37963	24817	41595	6918

Table 2: *Construction time (in seconds) of all experimented algorithms on two text collections: the Reuters corpus and the Amino-acid data set. N is the size of the text collection in bytes. The symbol '–' indicates that the test was stopped after* 63 *hours.*

The Reuters corpus					
N	BGS	Doubl	Doubl+Disc	Doubl+Disc+Radix	L-pieces
1324350	120/7865	2349/256389	2242/199535	4872/377549	837/57282
2578790	317/20708	4517/500151	4383/395018	10075/787916	1693/177003
5199134	929/60419	9095/1009359	8916/809603	22466/1761273	3386/360210
10509432	4347/282320	18284/2041285	18126/1655751	47571/3728159	6849/730651
20680547	14377/933064	35935/4017664	35904/3293234	96292/7550794	14243/1530995
26419271	24185/1568947	45911/5132822	45842/4202902	129071/10001152	18178/1956557
The Amino-Acid Test					
26358530	24181/1568773	41709/4656578	39499/3539148	105956/8222236	16118/1719883

Table 3: *Number of I/Os (bulk/total) of all experimented algorithms on two text collections: the Reuters corpus and the Amino-acid data set. N is the size of the text collection in bytes;* 64 *disk pages form a bulk-I/O.*

Results for the Manber-Myers' algorithm. It is not astonishing to observe that the construction time of MM-algorithm is outperformed by every other algorithm studied in this paper as soon as the working space exceeds the memory size. This worse behavior is due to the fact that the algorithm accesses the suffix array in an unstructured and unpredictable way. In fact its paging activity almost crashes the system, as we monitored by using the Solaris-tool *vmstat*. The vmstat value *sr*, which monitors the number of page scans per second, was constantly higher than 200. According to the Solaris system guide, this indicates that the system is constantly out of memory. Looking at Table 1, we infer that MM-algorithm should be chosen only when the text is shorter than $M/8$. In this case, the data structures fit into internal memory and thus the disk is never used. In our experimental setting, this actually happens for $N \leq 8$ Mbytes. When $N > 8$ Mbytes, the time complexity of MM-algorithm is still *quasi-linear* but the constant hidden in the big-Oh notation is very large due to the paging activity, thus making the algorithmic behavior unacceptable.

Results for the BaezaYates–Gonnet–Snider's algorithm. As observed in Section 3.2, the main *theoretical* drawback of this algorithm is the *cubic worst-case* complexity; but its small working space, its regular pattern of disk accesses and the small constants hidden in the big-Oh notation has lead us in the previous sections to think favorably of BGS for practical uses. Moreover, we conjectured a quadratic

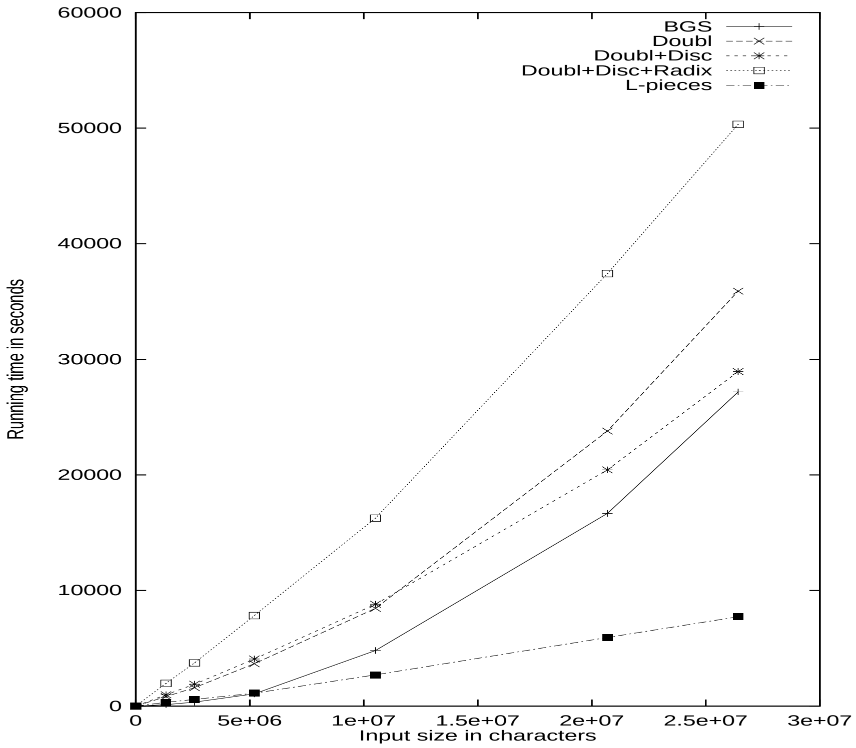

Figure 2: *Run-time of all construction approaches on the Reuters corpus.*

I/O-behavior in practice because of the *short* repeated substrings which usually occur in real texts. Our experiments show that we were right in all these suppositions. Indeed, if we double the text size, the running time increases by nearly a factor of four (see Table 2), and the number of bulk and random I/Os increase quadratically (see Table 3). The number of total and bulk I/Os is nearly identical for all data sets (Reuters, Amino-Acid and Random, see Table 3 and 5), so that the *practical behavior* is actually quadratic. Furthermore, it is not astonishing to verify experimentally that BGS is *faster* than any Doubling variant on the Reuters corpus and the Amino-Acid data set (see Figure 2). Consequently, it turns out to be the fastest algorithm for building a (unique) suffix array when $N \leq 25$ Mbytes. This scenario probably remains unchanged for text collections which are slightly larger than the ones we experimented in this paper; after that, the quadratic behavior of BGS will be probably no longer *"hidden"* by its nice algorithmic properties (largely discussed in Section 3.2).

In Table 3, we notice that (i) only the 1% of all disk accesses are random I/Os [13] (hence, most of them are bulk I/Os !); (ii) the algorithm performs the least number of random I/Os on both the data sets when building a unique suffix array; (iii) BGS is the fastest algorithm to construct one unique suffix array, and it is the second fastest algorithm in general. Additionally, we observe that the quadratic CPU-time complexity of the BGS-algorithm heavily influences (i.e., slows down) its efficiency and thus I/O is not the only bottleneck.

In summary, the BGS-algorithm is amazingly fast on *medium-size* text collections, and remains reasonably fast on larger data sets. It is not absolutely the fastest on larger and larger text collections because of its quadratic CPU- and I/O-complexities. Nonetheless, the small working space (possibly $4N$ bytes via a tricky implementation) and the ease of programming make the BGS-algorithm very appealing for software developers and practitioners, especially in applications where the space issue is the primary concern.

[13]Random I/Os are computed by means of the following formula: $IO_{random} = 1 - ((IO_{bulk} * 64)/IO_{total})$.

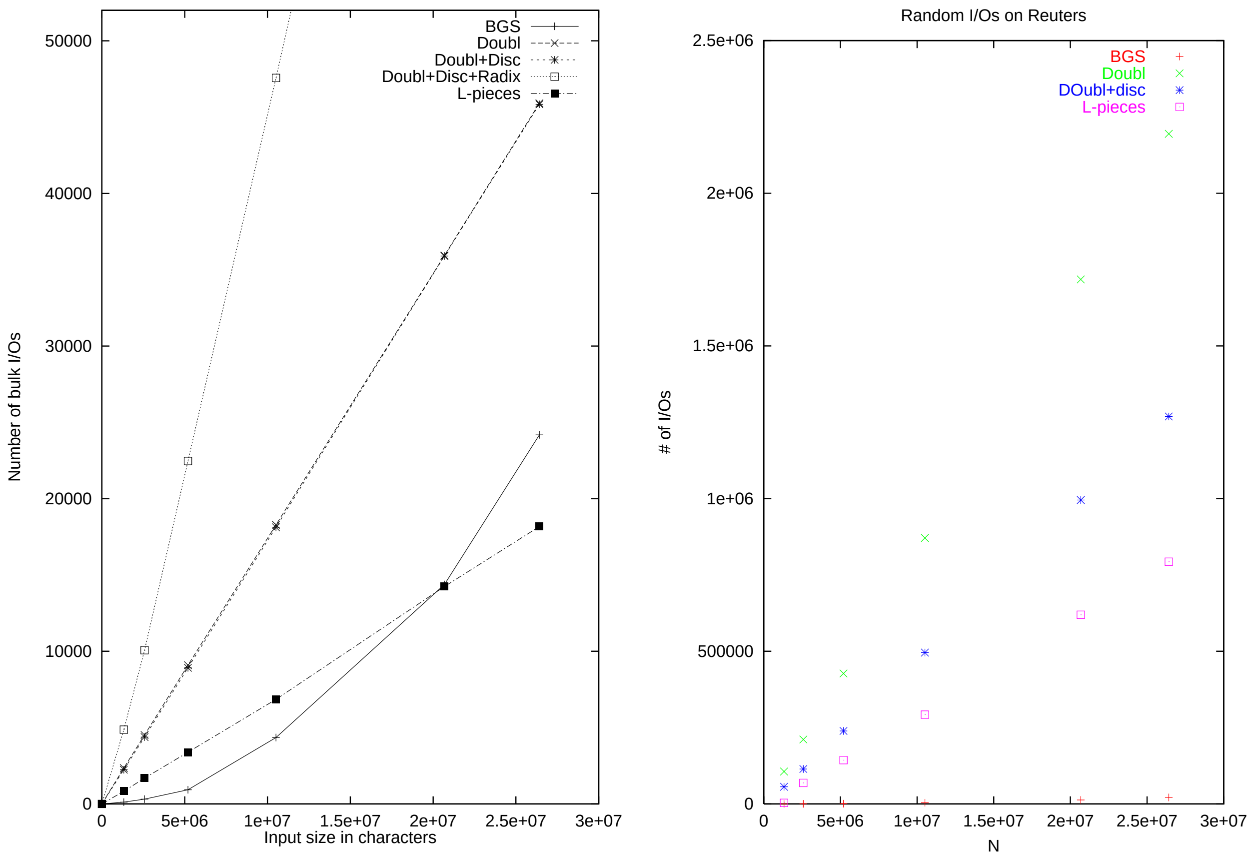

Figure 3: *Bulk and Random I/Os for all construction approaches. The bulk size is* 64 *disk pages.*

Results for the Doubling algorithm. The doubling algorithm performs 11 stages on the Reuters corpus, hence it performs 21 scans and 21 sorting steps on N tuples, where N is the text size. Consequently, we can conclude that there is a repeated substring in this text collection of length about 2^{12} (namely we detected a duplicated article). Figure 2 shows that the Doubling algorithm scales well in the tested input range: [14] If the size of the text doubles, the total running time doubles too. Among all disk accesses the 41% of them are random, and the number of bulk and random I/Os is larger than the ones executed by every other tested algorithm, except the Doubl+Disc+Radix variant which has higher worst-case complexity but smaller working space (see Table 3 and Section 4). Due to the high number of random I/Os and to the large working space (see Table 1), the Doubling algorithm is expected to surpass the performance of BGS only for *very large* values of N.

In summary, although theoretically interesting and almost asymptotically optimal, the Doubling algorithm is not appealing in practice; this motivated our development of the two variants discussed in Section 4 (namely, Doubl+Disc and Doubl+Disc+Radix algorithms).

Results for the Doubl+Disc algorithm. If we add the discarding strategy described in Section 4.1 to the Doubling algorithm, we achieve better performances as conjectured. The gain induced by the discarding step is approximately the 32% of the original running time both for the Reuters corpus and

[14]Notice that the curve for Doubling is not always linear. There is a "jump" in the running time as soon as the input has size $21 \cdot 10^6$. This is due to the system's pager-daemon that is responsible for managing the memory-pages. During the merge-step of multiway-mergesort, the number of free pages falls below a given threshold and the pager-daemon tries to free some memory pages. This goal can't be achieved because the mergesort constantly loads new blocks into memory. Therefore, this pager process runs almost all the time and increases the CPU-time of the merge-step of a factor of two.

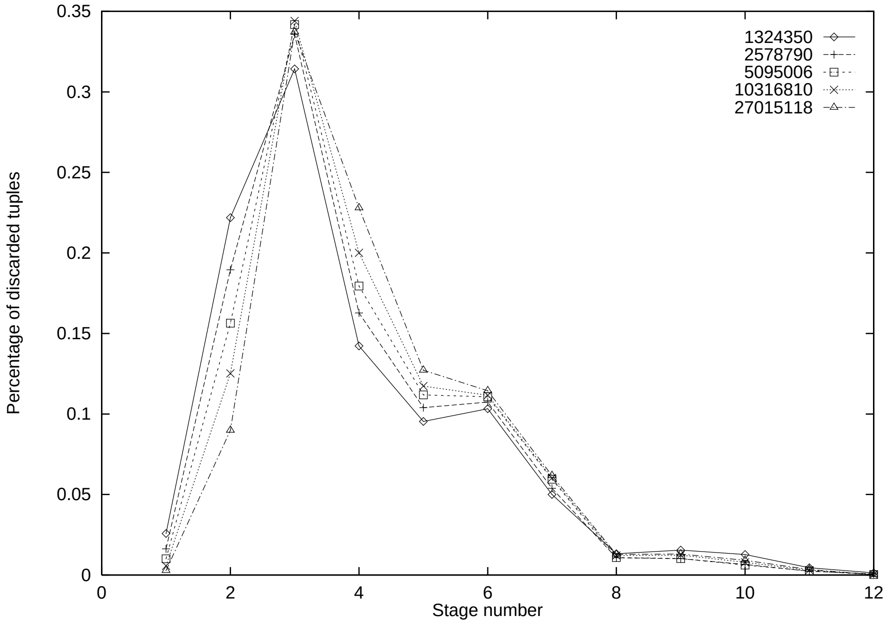

Figure 4: *The percentage of discarded tuples at each stage of the Doubl+Disc algorithm on the Reuters corpus.*

the Amino-acid data set. If we look in detail at the number of discarded tuples (see Figure 4), we see that for the Reuters corpus, this is small in the first two stages, while it becomes significant in stages three and four, where nearly 55% of the tuples are thrown away. Since we double the length of the substrings at each stage and we use the compression scheme of Section 3.3, we can infer that the Reuters corpus has a lot of substrings of length $16 \div 32$ that occur once in the collection but their prefixes of length $8 \div 16$ occur at least twice. We also point out that the curve indicating the number of discarded tuples is nearly the same as the size of the test set increases. This means that the number of discarded tuples is a "function" of the structure of the indexed text. For our experimental data sets, we save approximately 19% of the I/Os compared to Doubling. The percentage of random I/Os is 28%, this is much less than Doubling (42%), and hence discarding helps in reducing mainly the random I/Os (see also Table 3). The saving induced by the discarding strategy is expected to pay much more on larger text collections, because of the significant reduction in the number of manipulated tuples at the early stages, which should facilitate caching and prefetching operations. Consequently, if the time complexity is a much more important concern than the space occupancy, the Doubl+Disc algorithm is definitively the choice for building a (unique) *very large* suffix array.

Results for the Doubl+Disc+Radix algorithm. This algorithm is not as fast as we conjectured in Section 4.2, even for the tuple ordering of Step 2. The reason we have drawn from the experiments is the following. Notice that radix heaps are integer priority queues, and thus we cannot keep the parameter V small (to exploit Radix Heaps properties) by defining the first two components of a tuple as its priority. Hence, the sorting in Step 2 must be implemented via two sorting phases and this naturally *doubles* the work executed in this step. Additionally, the compression scheme of Section 3.3 cannot be used here because it would increase V too much in the early stages. Hence, the algorithm performs two more stages than Doubl-algorithm, and furthermore it does not take advantage of the discarding strategy because the number of discarded tuples in this two initial stages is very small. This way, it is not surprising to observe in Table 3 that the Doubl+Disc+Radix algorithm performs *twice* the I/Os of the other Doubling variants, and it is the slowest among all the tested algorithms. Hence the *compensation* between the number of discarded tuples and the increase in the I/O-complexity, conjectured in Section 4.2, do not actually arises in practice.

In summary, the Doubl+Disc+Radix algorithm can be interesting only in the light of its space requirements. However, if we compare space vs. time trade-off we can reasonably consider the Doubl+Disc+Radix algorithm *worse* than the BGS-algorithm because the former requires larger working space and it is expected to surpass the BGS-performance only for *very large* text collections (see Section 9 for further comments).

Results for the L-pieces algorithm. We fixed $L = 4$, used multi-way mergesort for sorting short strings and the Doubl-algorithm for constructing SA_L (see Section 4.3). Looking at Table 3 we notice that 40% of the total I/Os are random, and that the algorithm executes slightly more I/Os than the BGS-algorithm. Nonetheless, as shown in Figure 2 this algorithm is the fastest one. It is three to four times faster than BGS (due to its quadratic CPU-time) and four times faster than the Doubl+Disc algorithm (due to the larger number of I/Os). The running time distributes as follows: 63% of the overall time is used to build the compressed suffix array SA_L; the sorting of the short strings required only 4% of the overall time; the rest is used to build the other three suffix arrays. It must be said that for our test sizes, the short strings fit in internal memory at once thus making their sorting stage very fast. However, it is also clear that sorting short strings takes no more time than the one needed by *one* stage of the Doubl-algorithm. Furthermore, the construction of the other three suffix arrays, when executed entirely on disk, would account for no more than 1.5 stages of the Doubl-algorithm. Consequently, even in the case where the short strings and the other three suffix arrays reside on the disk, it is not hazardous to *conjecture* that this algorithm is still significantly faster than all the other approaches. Its only "drawback" is that it constructs the suffix array in four distinct pieces. Clearly, if the underlying text retrieval applications does not impose to have *one unique* suffix array, then this approach turns out to be de-facto *'the'* choice for constructing such a data structure (see Section 9 for further discussions).

Comparison of all construction approaches. We first compare all algorithms with respect to their time performances for increasing text lengths (see Figure 2). It is obvious from the discussions above that the MM-algorithm is the fastest one when the suffix array can be built entirely in internal memory. As soon as the working space crosses the 'memory border', there are various possible choices. The L-pieces algorithm is the natural choice whenever the splitting of the suffix array does not prevent its use in the underlying application. If instead a unique suffix array is needed, then the choice lies between the BGS-algorithm and the Doubl+Disc algorithm. For text collections which are not very big, the BGS-algorithm is preferable: it offers fast performance and very small working space. For larger text collections, the choice depends on the primary resource to be minimized: time or space ? In the former case, the Doubl+Disc algorithm is the winning choice; in the latter case, the BGS-algorithm is the best.

The Doubl+Disc+Radix variant could be interesting on huge text collections but this consideration cannot be claimed for sure because when N grows, the internal-memory size is also expected to grow ! Consequently, although the idea of using a space-efficient external-memory heap to implement the sorting step is valuable, a different heap structure should be devised to efficiently cope with large integer priorities and efficient I/O-performance. (This topic will be addressed in Section 9.)

With respect to the ease of programming, the BGS-algorithm seems still the best choice, unless the software developer has a library of general external sorting routines, in which case all Doubling variants turns out to be simple too.

6.1 Experiments on random data

The previous section left open three important questions:

1. How do the structural properties of the indexed text influence the performance of the Doubl+Disc algorithm ? Do these properties decrease the number of stages, and thus significantly improve its

overall performance ? How big is the improvement induced by the discarding strategy on more structured texts ?

2. What happens to the BGS-algorithm if we increase the text size ? Is its practical behavior "independent" of the text structure ?

3. What happens to the L-pieces algorithm when the four suffix arrays reside on disk ? Is this algorithm still significantly faster than all the other ones ?

In this section we will answer *positively* to all these questions by performing an extensive set of experiments on *three* sets of textual data which are *randomly and uniformly* drawn from alphabets of size 4 (random-small), size 16 (random-middle) and size 64 (random-large). For each alphabet size, we will indeed generate *two* text collections of 25 Mbytes and 50 Mbytes.

The choice of "randomly generated data sets" is motivated by the following two observations. By varying the alphabet size we can study the impact that the *average length* of the repeated substrings has on the performance of the discarding strategy. Indeed, the smaller is this length, the larger should be the number of tuples which are discarded at earlier stages, and thus the bigger should be the speed-up obtained by the Doubl+Disc algorithm. The experiments carried out on the Reuters corpus (see Section 6) did not allow us to complete this analysis because of the structural properties of this text collection. In fact, the Reuters corpus represents a pathological case because it has many *long* repeated text-substrings and this is usually untypical for natural linguistic texts. This was the reason why we conjectured at the end of Section 6 a much larger gain from the discarding strategy on more structured texts. An early validation of this conjecture was provided by the experiments carried out on the Amino-acid data set (see Table 2). Now we expect that the random data collections will be a good test-bed for providing further evidence. The same thing can be said about the BGS-algorithm whose "independent behavior from the structure of the indexed text in practice", conjectured in Section 6, can now be tested on *variously structured* texts.

Second, by varying the size of the indexed collection we can investigate the behavior of the L-pieces algorithm when the ordering of the short strings and the construction of the suffix arrays SA_1, SA_2, SA_3 operate directly on the disk (and not in internal memory). We can also test if the average length of the repeated substrings can influence the performance of $\mathcal{A}_{sa}$ when building SA_4. We notice that the construction of the other three suffix arrays is clearly not influenced by the structure of the underlying text, because it consists of just three sorting steps executed on a sequence of integer triples. In Section 6 we conjectured that a larger text collection should not significantly influence the overall performance of the L-pieces algorithm because, apart from the construction of SA_4, all the other algorithmic steps account for 2.5 stages of the Doubl-algorithm. Consequently, the overall work should be always much smaller than the one executed by all the Doubling variants. In what follows, we will validate this conjecture by running the L-pieces algorithm on larger data sets.

Results for the BGS-algorithm. It may appear surprising that BGS is the *slowest* algorithm on the random texts, after its successes on real text collections claimed in Section 6. It is more than twice slower than Doubl+Disc and up to nine times shower than L-pieces. However, nothing strange is going on in these experiments on random data because if we compare Table 5 to Table 3, we notice two things: (i) the number of bulk and total I/Os executed by BGS do not depend on the alphabet size and they are almost identical to those obtained on the Reuters corpus; (ii) the execution time of BGS decreases as the alphabet size grows, and it is smaller than the time required on the Reuters corpus (when $N \approx 25$ Mbytes, see also Table 4). The former observation implies that our conjecture on the " quadratic I/O-behavior in practice" is true, and in fact if we double the text size, the total number of I/Os increases by a factor of approximately four. The latter observation allows us to conclude that the CPU-time of BGS is affected by the length of the repeated substrings occurring in the data set. As the alphabet size increases, this length decreases on the average, and in turn decreases the running time of BGS. Such a dependence also shows that *both* I/O and CPU-time are significant bottlenecks for BGS, as pointed out already in Section 6.

Running times on the random texts			
N	L-pieces	Doubl+Disc	BGS
Alphabet Size 4			
25000000	4133	14130	21485
50000000	8334	34599	72552
Alphabet Size 16			
25000000	3838	11011	16162
50000000	7753	26450	62001
Alphabet Size 64			
25000000	3400	10080	15195
50000000	6802	25606	58417

Table 4: *Construction time (in seconds) required for random texts built on alphabet-sizes* 4, 16 *and* 64.

I/Os (bulk/total) on the random texts			
N	L-pieces	Doubl+Disc	BGS
Alphabet Size 4			
25000000	9466/970256	20661/2068791	22347/1449884
50000000	18912/1942979	41418/4143990	85481/5543929
Alphabet Size 16			
25000000	8499/860120	15320/1518126	22347/1449884
50000000	16984/1722707	30738/3042700	85481/5543929
Alphabet Size 64			
25000000	7531/749984	14643/1434118	22347/1449884
50000000	15056/1502435	30375/2996868	85481/5543929

Table 5: *Bulk and total I/Os required for random texts built on alphabets of size* 4, 16 *and* 64.

Results for the Doubl+Disc algorithm. The algorithm performs 4 stages on alphabet size 4, and 3 stages on alphabet sizes 16 and 32. Therefore, the random test with alphabet size 4 is the *worst case* for Doubl+Disc. In fact, the gain of the first two stages is negligible ($0.8 \cdot 10^{-8}$% discarded tuples); in the third stage, 98% of the tuples are discarded; the rest of the tuples is thrown away in the last fourth stage. This gives us the following insight: There are many *different* text substrings of 32 characters (i.e., 98%) whose 16-length prefix occurs at least twice in the collection. This validates our conjecture that the Doubl+Disc performance heavily depends on the average length of the repeated text-substrings.

If we look at the results on alphabet size 64, we see that the gain of the first stage is much bigger (about 5%), whereas the second stage throws away about 94% of the tuples. Consequently, for increasing alphabet sizes (approaching natural texts), Doubl+Disc gets faster and faster. Apart from the Reuters corpus, which seems indeed a pathological case, we therefore expect a much better performance on natural (and more structured) texts. So that we suggest its use in practice in place of the (plain) Doubling algorithm.

At this point it is worth to notice that the former two stages of Doubl+Disc do not discard any significant number of tuples (like on Reuters). Looking carefully to their algorithmic structure, we observe that these stages execute more work than the one required by the corresponding stages of Doubling because of Step 4 (Section 4.1). We experimentally checked this fact by running the Doubling algorithm on the random data sets and verifying an improvement of a factor of two ! Clearly, in the early two stages, the algorithm compares short substrings of length $1 \div 16$, and therefore it is unlikely that those substrings occur only once in a long text (and can be then discarded). Consequently, an insight coming from these experiments is that a *tuned* Doubl+Disc should follow an *hybrid* approach to gain the highest advantage from both the doubling and the discarding strategies: (plain) Doubling executed in the early (e.g. two) stages, Doubl+Disc for the next stages.

Results for the L-pieces algorithm. We again fix $L = 4$ and use the Doubling algorithm to construct SA_4. Using bigger texts (up to 50 Mbytes), we are ensured that all the steps of this algorithm operate on disk. The running time distributes as follows: 6% is used to sort short strings, 37% is used to build SA_4, and 47% is used to build the other three suffix arrays (via multiway mergesort). [15] Comparing Table 2 to Table 4, we conclude that the ordering of the short strings remains fast even when it is executed on disk, and it will never be a bottleneck. Moreover, the time required to build SA_4 clearly depends on the length of the longest repeated substring (see comments on Doubling), but $\mathcal{A}_{sa}$ is executed on a *compressed* text (of size $N/4$) where that length is reduced by a factor 4. Consequently, the L-pieces algorithm is usually between 2.9 and 8.3 times faster than Doubl+Disc (see Table 4). This speed-up is larger than the one we observed on the Reuters corpus (see Section 6), and thus validates our conjecture that a bigger text collection does not slow down the L-pieces algorithm.

6.2 Concluding remarks on our experiments

We first compare all experimented algorithms with respect to their time performance for increasing text lengths (see Table 2 for a summary). It is obvious from the previous discussions that the MM-algorithm is the fastest one when the suffix array can be built entirely in internal memory. As soon as the working space exceeds the memory size, we can choose among different algorithms depending on the resource to be minimized, either *time* or *space*. The L-pieces algorithm is the obvious choice whenever the splitting of the suffix array does not prevent its use in the underlying application. It is *more than* 3 times faster on the Reuters corpus than any other experimented algorithm; it is *more than* twice faster than the best Doubling variant on random texts. If, instead, a unique suffix array is needed, the choice depends on the structural properties of the text to be indexed. In presence of long repeated substrings, BGS is a good choice till *reasonably large* collections. For *very large* text collections or short repeated substrings, the hybrid variant of the Doubl+Disc algorithm is definitely worth to be used.

If the space resource is of primary concern, then BGS is a very good choice till reasonably large text collections. For huge sizes, Doubl+Disc+Radix is expected to be better in the light of its asymptotic I/O- and CPU-time complexity. However if one is allowed to keep the suffix array distributed into pieces, then the best construction algorithm results definitely the L-pieces algorithm: It is both the fastest and the cheapest in term of space occupancy (it only needs $6N$ bytes).

We wish to conclude this long discussion on our experimental data and tested algorithms by making a further, and we think necessary, consideration. The *running time* evaluations indicated in the previous tables and pictures are not clearly intended to be definitive. Algorithmic engineering and software tuning of the C++-code might definitively lead to improvements without anyway changing the algorithmic features of the experimented algorithms, and therefore without affecting significantly the scenario that we have depicted in these pages. Consequently, we feel not confident to give an *absolute quantitative* measure of the time performance of these algorithms in order to claim which is the "winner". There are too many system parameters (M, buffer size, cache size, memory bandwidth), disk parameters (B, seek, latency, bandwidth, cache), and structural properties of the indexed text that heavily influence those times. Nevertheless, the *qualitative* analysis developed in these sections should, in our opinion, safely route and clarify to the software developers which is the algorithm that best fits their wishes and needs.

We conclude our paper by addressing two other issues. The former concerns with the problem of building word-indexes on large text collection; we show in the next section that our results can be successfully applied to this case too without any loss in efficiency and without compromising the ease of programming so to achieve a uniform, simple and efficient approach to both the two indexing models. The latter issue is related to the intriguing, and apparently counterintuitive, "contradiction" between the effective practical performance of the BGS-algorithm and its unappealing worst-case behavior. In

[15] An attentive reader might observe the the distribution of the time only sums up to 90% of the total construction time. The missing 10% is required to copy back the computed suffix array.

Section 8, we deeply study its algorithmic structure and propose a new approach that follows its *basic philosophy* but in a significantly different manner, thus resulting in a novel algorithm which combines good practical qualities with efficient worst-case performances.

7 Constructing word-indexes

By using a simple and efficient preprocessing phase, we are also able to build a suffix array on a text where only the beginning of each word is indexed (hereafter called *word-based* suffix array). Our idea is based on a proposal of Andersson *et al.* [3] which was formulated in the context of suffix trees. We greatly simplify their presentation by exploiting the properties of suffix arrays. The preprocessing phase consists of the following steps:

1. **Scan** the text T and define as *index points* the text positions where a non-alphabetic character is followed by an alphabetic one.

2. Form a sequence S of strings which correspond to substrings of T occurring between consecutive *index points.*

3. **Sort** the strings in S via multiway mergesort.

4. Associate with each string its *rank* in the lexicographically ordered sequence, so that given $s_1, s_2 \in S$: if $s_1 = s_2$ then $name(s_1) = name(s_2)$, and if $s_1 <_L s_2$ then $name(s_1) < name(s_2)$.

5. **Sort** (backwards) S according to the starting positions in the original text T of its strings.

6. Create a compressed text T' via a simultaneous **scan** of T and (the sorted) S. Here, every substring of T occurring in S is replaced with its *rank.*

The symbols of T' are now *integers* in the range $[1, N]$. It is easy to show that the *word-based* suffix array for T is exactly the same as the suffix array of T'. Indeed, let us consider two suffixes $T[i, N]$ and $T[j, N]$ starting at the beginning of a word. They occur also in T' in a *compressed form* which preserves their lexicographic order because of the naming process. Consequently, the lexicographic comparison between $T[i, N]$ and $T[j, N]$ is the same as the one among the corresponding compressed suffixes of T'.

The cost of the preprocessing phase is dominated by the cost of sorting the string set S (step 3). As the average length of an English word is *six* characters [10], we immediately obtain from [1] (model A, strings shorter than B) that the I/O complexity of step 2 is $\Theta(N/B \log_{M/B} N/B)$, where N is the total number of string characters. Multiway mergesort is therefore an optimal algorithm to solve the string sorting problem in step 3. In general, we can guarantee that each string of S is always shorter than B characters by introducing some *dummy index points* that split the long substrings of T into *equal-length* shorter pieces. This approach does not suffer from the existence of very long repeated substrings that was reported by the authors of [26] in their quicksort/mergesort–based approaches; and therefore it is expected to work better on very large texts. With respect to [3], our approach does not use *tries* to manage S's strings and thus it reduces the overall working space and does not incur in the very well-known problems related to the management of unbalanced trees in external memory. Finally, since the preprocessing phase is based on sorting and scanning routines, we can again infer that this approach scales well on multi-disk and multi-processor machines, as we have largely discussed in the previous sections.

From the experiments executed on the L-pieces algorithm, we know that step 3 and step 5 above will be fast in practice. Furthermore, the compression in step 6 reduces the length of the repeated substrings in T, so that Doubl+Disc is expected to require very few stages when applied on T'. Consequently, we can expect that constructing word-indexes via suffix arrays is effective in real situations, and can benefit a lot from the study carried out in this paper.

8 The new BGS-algorithm

The experimental results of the previous sections have lead us to conclude that the BGS-algorithm is attractive for software developers because requires only $4N$ bytes of working space, it accesses the disk in a sequential manner, and is very simple to be programmed. However, its worst-case performance is poor and thus its real behavior is not well predictable but heavily depends on the structure of the indexed text. This limits the broad applicability of BGS, making it questionable at theoretical eyes.

In this section, we propose a new algorithm which deploys the *basic philosophy* underlying BGS (i.e., very long disk scans) but in a completely different manner: *the text T is examined from the right to the left.* The algorithm will choreograph this new approach with some additional data structures that allow to perform the suffix comparisons using only the information *available* in internal memory, thus avoiding the *random I/Os* in the worst case. The resulting algorithm still uses small working space (i.e. $8N$ bytes on disk), it is very simple to be programmed, it hides small constants in the big-Oh notation, and additionally it achieves effective *worst-case performance* (namely $O(N^2/M^2)$ worst-case bulk I/Os). This makes the practical behavior of the final algorithm *guaranteed on any* indexed text *independently of its structure*, thus overcoming the (theoretical) limitations of the BGS-algorithm, and still keeping its attractive practical properties.

Let us set $m = \ell M$, where $\ell < 1$ is a positive constant to be fixed later in order to guarantee that some auxiliary data structures can be fit into the internal memory. In order to simplify our discussion, let us also assume that N is a multiple of m, say $N = km$ for a positive integer k.

We divide the text T into k non-overlapping substrings of length m each, namely $T = T_k T_{k-1} \cdots T_2 T_1$ (they are numbered going from the right to the left, thus reflecting the "stage" when the algorithm will process each of them). For the sake of presentation, we also introduce an operator $\diamond$ that allows to go from *absolute positions* in T to *relative positions* in a text piece T_h. Namely, if $T[x]$ lies in the text piece T_h then $T[x] = T_h[x \diamond m]$, where $x \diamond m = ((x-1) \bmod m) + 1$.

The algorithm executes $\Theta(k) = \Theta(N/M)$ stages (like BGS) and processes the text *from the right to the left* (unlike BGS). The following invariant is kept inductively before stage h starts: *$S = T_{h-1} T_{h-2} \cdots T_1$ is the text part processed in the previous $(h-1)$ stages. The algorithm has computed and stored on disk the following two data structures: The suffix array SA_{ext} of the string S and its "inverse" array Pos_{ext}, which keeps at each entry $Pos_{ext}[j]$ the position in SA_{ext} of the suffix $S[j, |S|]$.* After all k stages are executed, we have $S = T$ and thus $SA = SA_{ext}$. See Figure 5 for an illustrative example.

The main idea underlying the *leftward*-scanning of the text is that when the h-th stage processes the text suffixes starting in T_h, it has already accumulated into SA_{ext} and Pos_{ext} some informations about the text suffixes starting to the right of T_h. This way, the comparison among the text suffixes starting in T_h will eventually exploit these two arrays, and thus use only *localized* information to eliminate *random I/Os*. The next Lemma formalizes this intuition:

Lemma 1 *A text suffix $T[i, N]$ starting into the substring T_h can be represented succinctly via the pair $(T[i, i+m-1], Pos_{ext}[(i+m) \diamond m])$. Consequently, all the text suffixes starting into T_h can be represented using overall $O(m)$ space.*

Proof: Text suffix $T[i, N]$ can be seen as the concatenation of two strings $T[i, i+m-1]$ and $T[i+m, N]$, where the second string is an arbitrary long text suffix. The position $i+m$ occurs in $T_{h-1} \cdots T_1$ $(= S)$ and in particular $T[i+m, N] = S[(i+m) \diamond m, |S|]$. This string can be succinctly represented with the number $Pos_{ext}[(i+m) \diamond m]$ which denotes its (lexicographic) position among S's suffixes. The space-bound easily follows by storing in internal memory the text substring $T_h\, T_{h-1}$ and the array $Pos_{ext}[2, m+1]$. ∎

Stage h preserves the invariant above and thus updates SA_{ext} and Pos_{ext} by *properly* inserting into them the information regarding the text suffixes which start in T_h. This way, the new SA_{ext} and Pos_{ext} will correctly refer to the *extended* string $T_h\, S$, thus preserving the invariant for the next $(h+1)$th stage (where $S = T_h\, S = T_h T_{h-1} \cdots T_2 T_1$). Details on the stage h follow (see Figure 5 below).

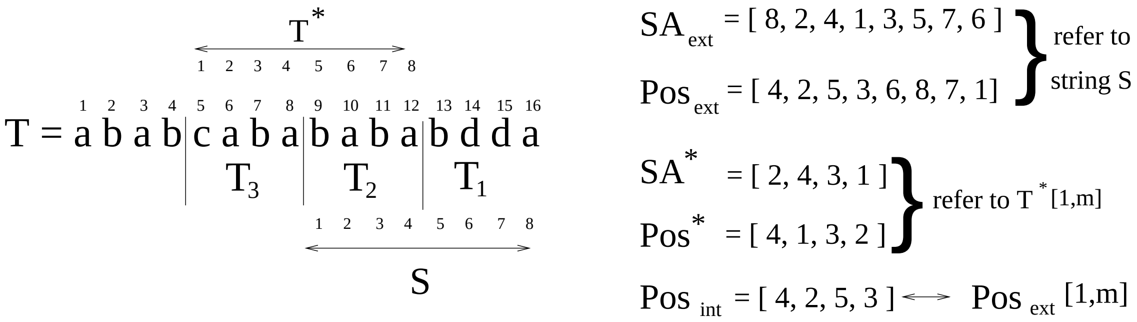

Figure 5: *The figure depicts the data structures used during stage $h = 3$, where $T^* = T_3 T_2$ and $S = T_2 T_1$. SA^* contains only the first m suffixes of T^*. Notice the order in SA^* of the two text suffixes starting at positions 2 and 4 of T^* (i.e., 6 and 8 of T). Restricted to their prefixes lying into T^*, these two suffixes satisfy the relation $4 \leq_L 2$, but considering them in their entirety (till the end of T), it is $2 \leq_L 4$. We can represent compactly $T[6,16]$ via the pair $\langle$ 'abab'$, 2\rangle$ and $T[8,16]$ via the pair $\langle$ 'abab'$, 3\rangle$; hence the comparison of those pairs gives the correct order and can be executed in internal memory (Lemma 1).*

1. Load T_h and T_{h-1} in the internal memory and set $T^* = T_h\, T_{h-1}$. ($O(1)$ bulk I/Os.)
2. Load the first m entries of Pos_{ext} into the array Pos_{int}. ($O(1)$ bulk I/Os.)
3. Construct the suffix array SA^* which contains the lexicographically ordered sequence of the text suffixes starting into T_h (recall that they extend till the end of T). This is done by means of three substeps which exploit the information kept in internal memory:
   (a) Build the suffix array of the string T^* using any internal memory algorithm (e.g. [24]). Then, throw away all suffixes of T^* which start in its second half. (No I/Os are required.)
   (b) Store the remaining m entries into SA^*. [16] (No I/Os are required.)
   (c) Refine the order in SA^* taking into account the suffixes in their entirety (i.e., considering also their characters outside T^*). Let $T^*[x, 2m]$ and $T^*[y, 2m]$ be two suffixes which lie adjacent into the current array SA^*, namely $SA^*[j] = x$ and $SA^*[j+1] = y$ for some value j, and such that one of them is the *prefix* of the other. Their order in SA^* may possibly be not correct (see Figure 5). Hence, the correct order is computed by comparing the two pairs $\langle T^*[x, x+m-1], Pos_{int}[(x+m) \diamond m]\rangle$ and $T^*[y, y+m-1], Pos_{int}[(y+m) \diamond m]$ (see Lemma 1). [17] This comparison is done for all the suffixes of T^* for which the ambiguity above arises. (No I/Os are required.)
4. Scan simultaneously the string S and the array Pos_{ext} ($O(h)$ bulk I/Os in total). For each suffix $S[j, |S|]$ do the following substeps:
   (a) Via a binary search, find the lexicographic position $pos(j)$ of that suffix into SA^* so that $S[j, |S|]$ follows the suffix of T starting at position $SA^*[pos(j) - 1]$ of T_h and precedes the suffix of T starting at position $SA^*[pos(j)]$ of T_h. Lemma 1 allows to perform the suffix comparisons of the binary search using only informations kept into internal memory.
   (b) Increment $C[pos(j)]$ by one unit.
   (c) Update the entry $Pos_{ext}[j] = Pos_{ext}[j] + pos(j) - 1$.

[16] Notice that this is not yet the correct SA^* because there might exist two text suffixes which start in T^* but have a common prefix which extends outside T^*. The next Step 3c takes care of this case without accessing the disk !

[17] This step is executed by employing the extra-space available into C and not yet used.

5. Build the new array $SA_{ext}[1, hm]$ by merging the old (external and shorter) array SA_{ext} with the (internal) array SA^* by means of the information available into $C[1, m+1]$. This requires a single disk scan (like BGS) during which the algorithm also executes the computation $SA_{ext}[j] = SA_{ext}[j] + m$, in order to take into account the fact that in the next $(h+1)$th stage the new string S has appended in front the text piece T_h. (Globally $O(h)$ bulk I/Os.)

6. Process in internal memory the array C as follows (no I/Os are executed):

   (a) Compute the Prefix-Sum of C.

   (b) Set $C[j] = C[j] + j$, for $j = 1, \ldots, m+1$.

   (c) Compute $Pos^*[1, m]$ as the inverse of SA^*, and then permute $C[1, m]$ according to $Pos^*[1, m]$. Namely, $C[i] = C[Pos^*[i]]$, simultaneously for all $i = 1, 2, \ldots, m$.

7. Build the new array $Pos_{ext}[1, hm]$ by appending $C[1, m]$ to the front of the current Pos_{ext}. ($O(h)$ bulk I/Os.)

The correctness of the algorithm immediately derives from Lemma 1 and from the following observations. As far as Step 6 is concerned, we observe that:

- Step 6a determines for each entry $C[j]$ the number of text suffixes which start in S and are *lexicographically smaller* than the text suffix starting at position $SA^*[j]$ of T_h.

- Step 6b keeps into account also the number of suffixes which start in T_h and are *lexicographically smaller* than the text suffix starting at position $SA^*[j]$ of T_h. These suffixes are $(j-1)$, so that the algorithm sums j to compute the final rank of that suffix (i.e., the one starting at $T_h[SA^*[j]]$) among all the suffixes of the string $T_h\, S$.

- Step 6c permutes the array C so that $C[j]$ gives the rank of the text suffix starting at $T_h[j]$ among all suffixes of the string $T_h\, S = T_h \cdots T_1$.

Since the string $T_h \cdots T_1$ corresponds to the string S of the next $(h+1)$th stage, we can conclude that Step 6 correctly stores in C the first m entries of the new array Pos^{ext} (Step 7). Finally, we point out that Step 4c correctly updates the entries of $Pos_{ext}[1, (h-1)m]$ by taking into account the insertion in SA_{ext} of the m text suffixes starting in T_h; and Step 5 correctly updates the entries of $SA_{ext}[1, (h-1)m]$ by taking into account the insertion in front of S of the m suffixes starting in T_h. In summary we can state the following result,

Theorem 1 *The suffix array of a text $T[1, N]$ can be constructed in $O(N^2/M^2)$ bulk-I/Os, no random-I/Os, and $8N$ disk space in the worst case. The overall CPU time is $O(\frac{N^2}{M}\log_2 M)$.*

The above parameter ℓ is set to fit SA^*, Pos^*, Pos_{int}, T^* and C into internal memory (notice that some space can be reused). We remark that the algorithm requires $4N$ bytes *more than* the space-optimized variant of the BGS-algorithm (see Section 3.2). Nonetheless, we can still get the $4N$ space-bound if we accept to compute only the array $Pos_{ext} = SA^{-1}$ (implicit SA). In any case, the overall working space is *much less* than the one required by all Doubling variants, and it is *exactly equal* to the one required by our implementation of the BGS-algorithm. The new BGS-algorithm is also very simple to be programmed and has small constants hidden in the big-Oh notation. Therefore, it preserves the nice algorithmic properties of BGS, but it now guarantees good worst-case performances. It should be consequently not astonishing for the reader if we conclude that the practical performance of the *new* BGS is surely *bounded above* by the interesting performance experimented for the classical BGS in Section 6.

9 Conclusions

It is often observed that practitioners use algorithms which tend to be different from what is claimed as optimal by theoreticians. This is doubtless because theoretical models tend to be simplifications of reality, and theoretical analysis need to use conservative assumptions. Our paper provided to some extent an example of this phenomenon—apparently *bad* theoretical algorithms are good in practice (see BGS and new-BGS). In the present paper we actually tried to *bridge* this difference by analyzing more deeply some suffix-array construction algorithms via the new accounting scheme of [12], taking more into account the specialties of current disk systems. This has lead us to a reasonable and significant taxonomy of all these algorithms. As it appears clear from the experiments, the final choice of the best algorithm depends on the available disk space, on the disk characteristics (which induce different trade-offs between random and bulk I/Os), on the structural features of the indexed text, and also on the patience of the user to wait for the completion of the suffix-array construction. The moral we draw from our experiments is that the design of an external-memory algorithm must take more and more into account the current technological trends, which boost interest toward the development of algorithms which prefer bulk rather than random I/Os because this paradigm can take advantage of the large disk-bandwidth and the high computational power of current computer systems.

The study and experiments so far conducted leads us to indicate some other directions of research that in our opinion deserve further investigation:

- Radix Heaps were chosen to reduce the space requirements of the doubling algorithm. Recently, in [8] has been presented a novel external-memory heap mainly based on an array topology, which requires strictly linear space and an optimal number of I/Os. The performance of this heap does not depend on the value and the type of the priorities stored into it, so that it has the space-advantages of the radix-heap but it overcomes its drawbacks. Therefore, it would be interesting to plug this new data structure into the Doubl+Disc algorithm and evaluate its time vs. space tradeoff.

- In [35], it is discussed a variant of the multiway mergesort which uses $NX + (N/B)\epsilon$ bytes to sort N items of size X bytes each ($\epsilon > 0$). This approach is I/O-optimal but the pattern of disk accesses is distributed randomly so that it is unclear how it might behave in real situations. It is therefore interesting to evaluate how this approach compares with the sorting routine above based on array-heaps [8].

- The algorithm presented in Section 4.3 is very efficient both in terms of I/Os and working space, but it produces $L = 4$ distinct suffix arrays. It would be interesting to establish how much it costs in practice to *merge* these four arrays into one unique suffix array. In this respect, it would be worth to use the *lazy-trie* data structure proposed in [1].

- Recently, Farach *et al.* [12] devised the first I/O-optimal algorithm for building *suffix trees*. Although asymptotically optimal (both in time and space), that algorithm uses more space than the algorithms discussed in this paper because it operates on a tree topology. It is not yet clear how this approach could be used to *directly* construct suffix arrays. This is an important problem both theoretically and experimentally.

- Our new algorithms scale theoretically well in a distributed environment where D disks or P workstations are available. However, there are only few practical experiments in this setting [26]. We hope in the future to extend our results to this high-performance environment.

References

[1] L. Arge, P. Ferragina, R. Grossi and J. S. Vitter. On sorting Strings in External Memory. *ACM Symposium on Theory of Computing*, pp. 540-548, 1997.

[2] R. Ahuja, K. Mehlhorn, J. B. Orlin and R. E. Tarjan. Faster Algorithms for the Shortest Path Problem. *Journal of the ACM*(2), pp. 213-223, 1990.

[3] A. Andersson, N. J. Larson and K. Swanson. Suffix Trees on Words. *Combinatorial Pattern Matching Conference*, LNCS 1075, pp. 102-115, 1996.

[4] A. Andersson and S. Nilsson. Efficient implementation of Suffix Trees. *Software Practice and Experience*, 2(25): 129-141, 1995.

[5] R. Baeza-Yates, E. F. Barbosa and N. Ziviani. Hierarchies of Indices for Text Searching. *Information Systems*, 21(6):497-514, 1996.

[6] R. Baeza-Yates, A. Moffat and G. Navarro. Searching Large Text Collections. In *Handbook of Massive Date Sets*, Kluwer, August 2000 (to appear).

[7] S. Burkhard, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals and M. Vingron. q-gram based database searching unsing a suffix array (QUASAR). *International Conference on Computational Molecular Biology (RECOMB)*, 1999.

[8] K. Brengel, A. Crauser, P. Ferragina and U. Meyer. An Experimental Study of Priority Queues in External Memory. *3rd International Workshop on Algorithm Engineering*, pp. 345–360, LNCS 1668, 1999.

[9] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *TR 124, Digital SRC Research Report*, 1994.

[10] D. R. Clark and J. I. Munro. Efficient Suffix Trees on Secondary Storage. *ACM-SIAM Symposium on Discrete Algorithms*, pp. 383-391, 1996.

[11] A. Crauser and K. Mehlhorn. LEDA-SM: Extending LEDA to Secondary Memory. *3rd International Workshop on Algorithm Engineering*, pp.228–242, LNCS 1668, 1999.

[12] M. Farach, P. Ferragina and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 2000 (to appear).

[13] C.L. Feng. PAT-Tree-Based Keyword Extraction for Chinese Information Retrieval. *ACM SIGIR*, pp. 50-58, 1997.

[14] P. Ferragina and R. Grossi. The String B-Tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, vol. 46(2), 236–280, March 1999.

[15] E. A. Fox, R. M. Akcyn, R. K. Furuta and J. J. Leggett. Special Issue: Digital libraries. *Communications of the ACM*, 38(4), 1995.

[16] T. Goodrich. Communication-efficient parallel sorting. *ACM Symposium on Theory of Computing*, pp. 247–256, 1996.

[17] G. H. Gonnet, R. A. Baeza-Yates and T. Snider. New indices for text:PAT trees and PAT arrays. In *Information Retrieval – Data Structures and Algorithms*, W.B. Frakes and R. BaezaYates Editors, pp. 66-82, Prentice-Hall, 1992.

[18] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*, Cambridge University Press, 1997.

[19] R. M. Karp, R. E. Miller and A. L. Rosenberg. Rapid identification of repeated patterns in strings, arrays and trees. *ACM Symposium on Theory of Computing*, pp. 125-136, 1972.

[20] J. P. Kitajima, G. Navarro, B. Ribeiro-Neto and N. Ziviani, Distributed Generation of Suffix Arrays: A Quicksort Based Approach. *South American Workshop on String Processing*, Carleton University Press,pp. 53–69, 1997.

[21] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching.* Vol. 3, Addison-Wesley Publishing Co. 1969.

[22] S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software – Practice and Experience*, 29(3):1149–1171, 1999.

[23] N. J. Larsson. Extended application of suffix trees to data compression. *IEEE Data Compression Conference*, 1996.

[24] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal of Computing 22*, 5,pp. 935-948, 1993.

[25] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM* 23(2):262-272, 1976.

[26] G. Navarro, J.P. Kitajima, B.A. Ribeiro-Neto and N. Ziviani. Distributed Generation of Suffix Arrays. *Combinatorial Pattern Matching Conference*, pp. 103–115, 1997.

[27] S. Näher and K. Mehlhorn. LEDA: A Platform for Combiantorial and Geometric Computing. *Cambridge University Press*, 1999.

[28] M. H. Nodine and J. S. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM*, pp. 919-933, 1995.

[29] Quantum Cooperation. Storage Technology and Trends. `http://www.quantum.com/src/tt/storage_tech_trends.htm`

[30] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, 1994.

[31] K. Sadakane. A Fast Algorithm for Making Suffix Arrays and for Burrows-Wheeler Transformation. *IEEE Data Compression Conference*, 1998.

[32] W. R. Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley Professional Computing Series, 1992.

[33] J. Vitter. External memory algorithms. Invited Tutorial, *ACM Symposium on Principles of Database Systems (PODS '98)*, Seattle, WA, 1998. Also Invited paper in *European Symposium on Algorithms (ESA '98)*, Venice, Italy, 1998.

[34] J. Zobel, A. Moffat and K. Ramamohanarao. Guidelines for presentation and comparison of indexing techniques. *SIGMOD Record 25*, 3:10–15, 1996.

[35] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes.* Van Nostrand Reinhold, 115th Avenue, New York, 1994.