String Search Experimentation Using Massive Data

Alistair Moffat and Simon Gog

Department of Computing and Information Systems,
The University of Melbourne, Australia

Descriptions of new string search or indexing algorithms are often accompanied by an experimental evaluation. In this article we provide guidance as to how such investigations can be carried out, drawing on our experience of measurement in this field. In particular, we describe methodologies for stratifying patterns according to their length and frequency, so that precise response-time measurements can be made; and we describe a metric for categorizing the extent of "repetitiveness" in a text, so that data set type can also be factored in to evaluations. We show that separating these concepts allows a greater understanding of the behavior of string search algorithms.

1. Introduction

When a new algorithm is presented, it is usual for it to be accompanied by either an asymptotic analysis, showing that it has certain behaviors in the limit; and/or an experimental evaluation, demonstrating that it has certain behaviors in practice. A successful experimental evaluation requires a number of key components: a set of data that is deemed to be plausibly representative of the type and scale of "real" data; an operation stream against it that is typical of a "real" operation stream; a state-of-the-art baseline mechanism that can be used as a credible reference point; and an evaluation methodology that provides confidence that the observed outcomes are consistent and repeatable. The latter aspect includes selection of appropriate hardware, selection of appropriate assessment metrics, and appropriate disclosure of all outcomes (including those that might be less than flattering) as part of the research dissemination process.

Our purpose in this article is to describe how these goals can be achieved in the area of string search algorithms. This is a field with a long history; a strong culture of experimental evaluation; a tradition of shared (that is, publicly available) data sets; and a tradition of shared publicly available software. But even with these various positive factors, it is not always clear how experiments should be structured. In reflecting on our own experiences in this area, we hope to provide a resource that other researchers will be able to build on as they plan further experimental evaluations.

2. Evaluating String Search

String search is a well-known problem: given a text $\mathsf{T}[0..n-1]$ over an alphabet Σ of size σ, and a pattern $\mathsf{P} = [0..m-1]$ also over Σ (with $m \ll n$), determine the number of times P appears in T (*count* queries) or the locations at which P appears (*locate* queries). A large range of algorithmic mechanisms have been described in the literature, falling in to two broad groups – those that make no assumption about T and P, and assume that every instance is different and hence that pre-computation can take place on P but not T; and those that assume that T is fixed, and that only P varies by instance. In the latter case, pre-computation costs involving T can be assumed to be amortized over many instances of P.

Defending a new algorithm via experimentation appears simple enough: it has to be implemented, some data selected, some timings made, and then a comparison against a baseline or reference system undertaken. Shared resources such as those provided by the Pizza & Chili Corpus[1] include a range of files, spanning different data types and sizes. Previous text collections have also been used in experimentation, including the Calgary Corpus and Canterbury Corpus[2], formed originally as resources for text compression evaluation. The datasets have grown steadily in size: the largest file in the 1987 Calgary Corpus was 750 kB; the largest English text file in the Large Canterbury Corpus [Arnold and Bell, 1997] was 3.8 MB; and the largest English text file in the Pizza & Chili Corpus is 2.05 GB. Other file types provided in these three collections include genetic data, XML data, program source code, and music pitch data. Larger datasets derived from web and other on-line sources are provided as part of the NIST-sponsored TREC project[3].

The task of selecting patterns comes next; and there are two typical approaches. The first is to gain access to a stream of user queries and apply them to the text. There are a number of ways this can be done: a group of users might agree to the queries that they issue being monitored as part of a research project; or an institution might grant access to a query log for one of its internal search services; or a commercial web company might be willing to release suitable anonymized data. It is then necessary for the researcher to argue that the data T that they propose to use is appropriate for the patterns P that they have collected. It may make little sense, for example, for queries harvested via a library catalog system (with a focus on author names and technical words) to be applied to, say, web data crawled from the `.com` domain.

The second approach is to generate patterns directly from the experimental text: for example, Navarro and Salmela [2009] write "the patterns are random substrings of the text [and] for each pattern length, 1,000 patterns were generated".

Table 1 analyzes the effect of random pattern selection when applied to files from the Pizza & Chili Corpus. To generate the table, strings of length $m \in \{8, 16, 32, 64\}$ were extracted at all $n - m$ starting locations in each file, and then categorized by frequency. Hence, the numbers in the table reflect the overall distribution from which random sampling "over the text" selects a subset. There is no great surprise in the trends evident in the mean values reported – shorter strings are likely to occur more often than longer ones. But the very large discrepancy between mean and median that arises for all but the longest of patterns highlights a problem of this approach – it is biased in favor of small numbers of frequently-occurring strings. In some cases this effect may be so pronounced that random pattern sets contain multiple instances of the same pattern. When the algorithm being tested is such that the generation of each answer involves computation time – that is, when the running time is a function of all of n, m, and k, where k is the number of answers, rather than of n and m alone – this bias means that "average" execution times may not be representative of true search costs.

[1] See `http://pizzachili.dcc.uchile.cl/`
[2] Both available at `http://corpus.canterbury.ac.nz/descriptions/`
[3] See `http://trec.nist.gov` for details.

File	$m = 8$		$m = 16$		$m = 32$		$m = 64$	
	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.
Sources-200	30,551	93	7,126	3	1,750	1	150	1
Proteins-200	267	2	125	2	48	1	24	1
DNA-200	8,144	5,104	186	1	10	1	2	1
English-200	2,707	149	156	2	37	1	3	1
English-2108	27,632	1,481	969	2	51	2	2	2
XML-200	146,763	7,563	38,822	287	7,766	3	128	1

Table 1. Occurrence statistics for randomly selected patterns. All values are rounded to the nearest integer.

Answers k	$m = 8$	$m = 16$	$m = 32$	$m = 64$
English-200				
$k = 1$	9,528,686	92,274,275	123,964,935	127,201,005
$k = 10$	966,344	712,414	22,835	17,103
$k = 100$	149,863	13,895	*1,349*	*727*
$k = 1{,}000$	13,956	*146*	*36*	*2*
English-2108				
$k = 1$	29,434,962	541,592,957	967,710,430	995,311,938
$k = 10$	4,681,780	13,072,884	957,290	625,774
$k = 100$	924,194	520,995	*4,336*	*1,345*
$k = 1{,}000$	149,741	12,714	*97*	*12*

Table 2. Size of stratified pattern universes (set *patts* in Figure 1) for the Pizza & Chili Corpus files English-200 and English-2108 when occurrence counts are allowed to vary from the target k by as much as $\pm 25\%$.

3. Selecting Patterns

Our first contribution is summarized in Figure 1, which describes how sets of random patterns stratified by pattern length m and approximate frequency k can be identified via a pre-order traversal of a suffix tree. In this mechanism, the *count* attribute of a node v in the suffix tree indicates the size of the subtree rooted at that node, and hence the total number of times that the string indicated by the v's *label* attribute appears in T; and the *depth* attribute indicates the length of the string required to reach that node from the root. Note that there is tension between the desire for accuracy in terms of the occurrence frequency bands (in the pseudo-code, $k \pm 25\%$) and the need to develop pattern sets of a certain size; the broader the banding on k, the more likely it is that the desired total of $num \times sze$ distinct patterns can be achieved. Note that while function *generate_patterns*() is presented as taking in scalar arguments m and k, it is also possible to take in sets of values $\mathbf{m}$ and $\mathbf{k}$, and in a single pass over the suffix tree, compute all $|\mathbf{m}| \cdot |\mathbf{k}|$ sets of patterns, one set for each $(m,k) \in \mathbf{m} \times \mathbf{k}$.

Table 2 shows the size of the set *patts* that is generated for a range of pattern lengths m and targets k (with a $\pm 25\%$ tolerance) for two of the Pizza & Chili Corpus files, English-200 and English-2108. With $num = 10$ and $sze = 1{,}000$ (the combination used in the experiments below), there are five pairings of m and k that result in insufficiently

```
$generate\_patterns(\mathsf{T}, n, m, k, num, sze)$
  // compute $num$ sets of patterns, each of size $sze$, with each pattern
     of length $m$ and having approximately $k$ matches in $\mathsf{T}[0..n-1]$
  construct a suffix tree $ST$ for $\mathsf{T}[0..n-1]$
  $patts \leftarrow \{\}$
  **for** each node $v$ in a pre-order traversal of $ST$ **do**
    **if** $count(v) < 0.75k$ **then**
      **continue** the traversal, but bypass all of the children of $v$
    **else if** $depth(v) \geq m$ **then**
      **if** $count(v) < 1.25k$ **then**
        $patts \leftarrow patts \cup \{$the first $m$ symbols of $label(v)\}$
      **continue** the traversal, but bypass all of the children of $v$
  **if** $|patts| < num \times sze$ **then**
    **return** "insufficient patterns found, generation not possible"
  **generate** (without replacement) $num$ random subsets of $patts$,
    each of size $sze$
  **return** the $num$ sets of patterns
```

Figure 1. Computing multiple sets of experimental patterns stratified by length m and occurrences k. The frequency boundaries $[0.75, 1.25]$ can be altered for different requirements.

many patterns being available on English-200, and four pairings on English-2108. Those combinations are shown in italics in the table. Note that the "insufficient patterns available" zone is not restricted to the region on the bottom right, and can occur anywhere in the table. For example, on the DNA-200 file, when $m = 8$ there are fewer than 10,000 available patterns for all of $k = 10$, $k = 100$, and $k = 1{,}000$, even allowing for the $\pm 25\%$ flexibility. Combinations for which the pattern generation process did not succeed in finding sufficiently many patterns are indicated by "—" entries in the results tables below.

The FM-INDEX [Ferragina and Manzini, 2005] was selected as an example indexing structure, and instances constructed for the set of Pizza & Chili Corpus test files listed in Table 1. Each FM-INDEX was then used to search for each pattern in each of the corresponding test sets (formed using $num = 10$ and $sze = 1{,}000$), performing both *count* queries and *locate* queries. The hardware used for the searching experiments was a MacBook Pro with 16 GB of primary memory, 6 MB L3 cache, and SSD secondary memory.

Using the notation of the `sdsl` succinct data structures library[4], the FM-INDEX was formed using:

```
csa_wt<
    wt_huff<
        bit_vector,
        rank_support_v5<>,
        select_support_scan<>,
        select_support_scan<>
        >,
    32, 32,
```

[4] `https://github.com/simongog/sdsl`

Answers	$m = 8$	$m = 16$	$m = 32$	$m = 64$
count queries				
Uncontrolled	7.8 ± 0.4	14.9 ± 0.6	28.3 ± 1.1	54.2 ± 2.8
$k = 1$	8.4 ± 0.3	14.8 ± 0.5	28.3 ± 0.9	53.6 ± 1.5
$k = 10$	8.2 ± 0.3	14.7 ± 0.5	27.9 ± 0.7	52.8 ± 2.3
$k = 100$	7.7 ± 0.5	14.5 ± 0.9	—	—
$k = 1{,}000$	7.7 ± 0.3	14.3 ± 0.7	—	—
locate queries				
Uncontrolled	$266{,}150 \pm 38{,}439$	$7{,}663 \pm 9{,}473$	210 ± 197	81.7 ± 14.5
$k = 1$	27.1 ± 3.0	31.3 ± 2.3	46.1 ± 3.6	70.9 ± 2.7
$k = 10$	126.1 ± 5.5	125.1 ± 4.6	84.5 ± 4.4	100.2 ± 2.9
$k = 100$	$1{,}068.3 \pm 37.8$	$1{,}045.1 \pm 32.1$	—	—
$k = 1{,}000$	$9{,}534.4 \pm 204.2$	$9{,}414.2 \pm 241.5$	—	—

Table 3. Time (milliseconds per set of 1,000 queries) to resolve *count* queries (top) and *locate* queries (bottom) against English-2108 using an FM-INDEX, plus the standard deviation measured over 10 such pattern sets.

```
text_order_sa_sampling<
    sd_vector<>
    >
>
```

that is, as a Huffman-shaped wavelet tree [Mäkinen and Navarro, 2005], using a 6.25%-overhead rank structure on uncompressed bitvectors, with every 32nd suffix sampled and marked using a compressed SD-bitvector. In addition, every 32nd inverse suffix array entry was sampled. This combination of options yields an FM-INDEX that preferences access speed over storage space; other versions are also possible that preference reduced storage space at the expense of speed. As is demonstrated by Gog and Petri [2013], the `sdsl` library provides the best available implementation of the remarkable FM-INDEX data structure. Experiments were sequenced in a cyclic manner, so that first pattern set for each m and k pairing was executed before all the second sets, and so on, to ensure that disruptions caused by intermittent background processes did not affect just one pairing.

Table 3 shows how the stratified query generation approach allows more accurate measurement of query times than does selection of random strings. Each entry represents the time (averaged over $num = 10$ experiments) taken to search for a set of $sze = 1{,}000$ patterns, and is followed by the standard deviation of the running times over the same $num = 10$ runs. In the case of the "uncontrolled" experiment, ten sets of 1,000 random locations were selected in the text, and the next m symbols taken to be the pattern. The top half of the table shows the cost of *count* queries; in the FM-INDEX structure these are executed in $O(m \log \sigma)$ time, and are independent of both n, the length of the string T, and of k, the number of answers. That second relationship is apparent in the table, and for these queries the "uncontrolled" pattern sets yield the same overall outcomes as the stratified pattern sets, with a slight decrease in execution time when patterns are common. In all of the *count* experiments, the standard deviation is between 4% and 7% of the measured time for 1,000 queries

The situation is quite different when *locate* queries are measured, shown in the bottom half of Table 3. These queries require $O(m \log \sigma + k)$ execution time using an FM-INDEX, and the high level of variability in the frequency of the patterns in the "uncontrolled" query set translates directly into high variance in measured execution times over sets of 1,000 patterns. That is, even taking 1,000 patterns at a time is not sufficient to smooth out the per-query variance. Indeed, in two of the uncontrolled cases the standard deviation is as large as the measured quantity, which suggests that the measurement cannot be relied on in any way. Having a high standard deviation in an experiment is particularly problematic if different implementations or algorithms are to be compared using a statistical test. The higher the standard deviation, the less likely it is that any given underlying measurement delta will be successfully identified. If the "uncontrolled" *locate* queries are used as the basis for an experiment comparing two techniques, a statistical comparison would in all likelihood fail to indicate significance.

The measured standard deviation for *locate* queries also increases for the stratified pattern sets, and now it is as much as 10% of the measured average when $k = 1$. It then decreases (as a fraction) as k becomes larger. This pattern of observations is consistent with the $\pm 25\%$ tolerance on k when the pattern sets are constructed. Tighter bounds on k would reduce the variance, but might mean that some pairings of k and m could no longer be run.

Some authors make use of "uncontrolled" pattern sets, and then divide the measured running time by the total number of answers k, to get a per-answer running time. That alternative normalization process can be anticipated, and embedded into the query selection mechanism. For example, Ferragina et al. [2008] write: "we locate sufficient random patterns of length 5 to obtain a total of 2 to 3 million occurrences per text". This approach meant that in their experiments just 10 queries were measured against the DNA-200 file, but 3,500 were used when testing on Proteins-200. The average "per answer" query times that are generated are relatively stable in terms of presenting them in graphs and tables, but make it difficult to apply statistical tests, since the effect of m is both variable, because of the different number of patterns used, and also masked by the division. Moreover, since the measured standard deviations are for whole pattern sets, they cannot be normalized the way measured per-answer means can be.

Fischer et al. [2008] also consider the process of identifying patterns according to their frequency in a text T. Compared to the process described here, their space utilization is likely to be the same, or even lower; but because of their emphasis on reducing the space cost, execution is likely to be substantially slower.

4. Measuring Repetitiveness

A second factor that affects the space requirement and query time behavior of an index is the amount of "repetitiveness" that is present, or the extent to which it can be compressed [Claude et al., 2011; Navarro, 2012; Sirén et al., 2008]. One standard way of measuring compressibility is to compute H_k, the kth order entropy of the input string – the weighted average of the entropies of the probability distributions of all symbols immediately following each of the as many as $\max\{\sigma^k, n-k\}$ different k-symbol sequences that occur in T. This suggests that $n\mathsf{H}_k + \sigma^k$ is an estimate of the size of the compressed representation, where the σ^k term approximates the aggregate cost of storing the probabilities associated with the set of kth-order contexts. Hence, if the cost of storing the probabilities is to be less than n, it can be assumed that $k \leq \log_\sigma n$.

Repetitiveness has forms other than the "predictability based on k prior symbols" that is captured by H_k. Consider a text T created by first generating a random string of length $n/2$ over σ, and then doubling it to make a string of length n. When $k < \log_\sigma n$, each k-symbol context occurs sufficiently many times in the first $n/2$ symbols that a uniform probability distribution over σ is correctly inferred, and $\mathsf{H}_k \approx \log_2 \sigma$. Doubling the length of the string does not alter that distribution. Hence, when k is small relative to $\log_\sigma n$, the $n\mathsf{H}_k$ component of the estimated cost of representing a doubled string also doubles. But a doubled (or tripled, or quadrupled) string should certainly be regarded as being highly repetitive.

As an alternative mechanism for quantifying repetitiveness, we propose the use of an index computed from LCP values, where $\mathsf{LCP}[i]$ is the length of the common prefix shared between the $i-1$ th and i th suffixes of the text when the set of all suffixes is ordered lexicographically (with $\mathsf{LCP}[0]$ defined to be zero):

$$\mathsf{R}(\mathsf{T}) = \frac{1}{n}\sum_{i=0}^{n-1} \frac{\log_2 n + \log_2(1+\mathsf{LCP}[i]))}{1+\mathsf{LCP}[i]}\,. \tag{4.1}$$

This computation is motivated by factor-based compression mechanisms, such as the two Lempel-Ziv families. The key idea is that, in the absence of a parsing regime to identify which factors are required, and in the absence of a particular coding regime to indicate how factor identifiers and lengths are stored, it is still possible to obtain a numeric estimate of the performance of such a compression scheme. In Equation 4.1, all possible factors are considered, and their cost versus benefit captured by computing an average over n, hence the $(1/n)\sum$ part of the computation. Within that average, if the i th suffix gets used as a factor, then it costs $\log_2 n$ bits or less to identify its starting point in T, and approximately $\log_2(1+\mathsf{LCP}[i])$ bits to describe its length. Moreover, if that i th factor is employed, it spans $\mathsf{LCP}[i]$ symbols. That is, Equation 4.1 estimates an average cost ratio over all n maximal-length factors in T; like H_k, it has units of "bits per symbol".

To show the behavior of R numerically, Table 4 lists the scores achieved for combinations of σ and n for random strings T, and for doubled and quadrupled strings. Since a randomly generated string over an alphabet of size σ has an average LCP that converges to $\log_\sigma n$ for large n, the expected value of Equation 4.1 converges to $\log_2 \sigma$ as n becomes large. At the other extreme, on a file contain n repetitions of a single symbol, the value of R is $O((\log^2 n)/n)$.

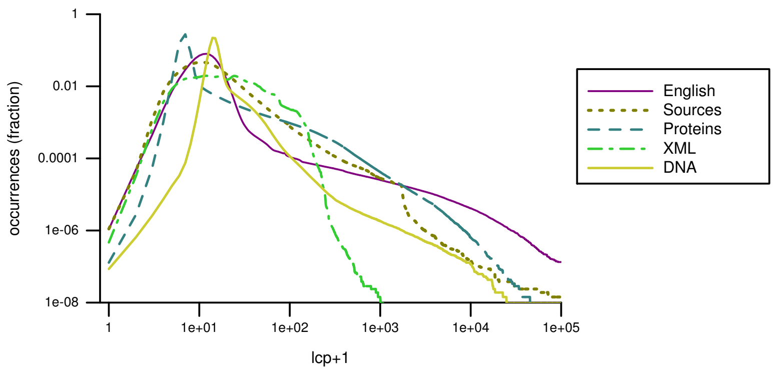
Figure 2 shows the LCP distribution across several of the files in the Pizza & Chili Corpus 200 MB collection, and several files from the Pizza & Chili Corpus repetitive collection[5]. Comparing the two graphs, there is a noticeable difference in the LCP profile; it is that difference that we seek to capture in Equation 4.1.

Figure 3 documents the relationship between H_k and R. A string that has a low H_k will also have a low R score, but the reverse is not required to be true. The set of Pizza & Chili Corpus highly repetitive files have a range of H_k values that overlaps with the regular 200 MB files, but all have much lower R scores. The two very low R scores are for the files Einstein-de and Einstein-en, which are change histories of the German and English wikipedia pages about Albert Einstein. Each of these files is a concatenation of multiple similar versions of the same file, and hence contains many long duplicate strings; as anticipated, their repetitiveness scores R are substantially less than their compressibility scores H_k.

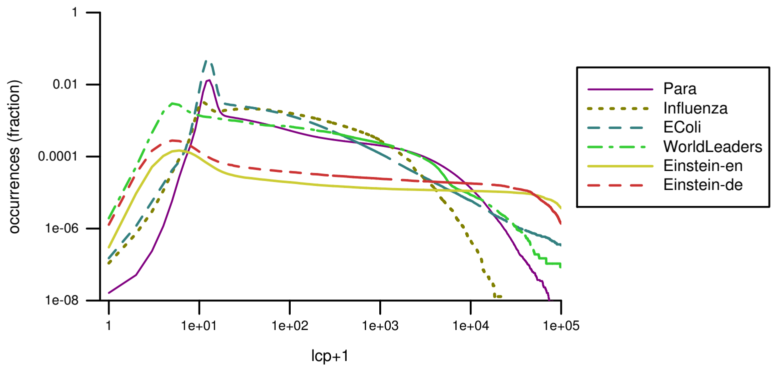
[5] `http://pizzachili.dcc.uchile.cl/repcorpus.html`

n	random T, $\sigma =$			doubled T, $\sigma =$			quadrupled T, $\sigma =$		
	4	16	64	4	16	64	4	16	64
100,000	2.36	4.27	5.94	1.24	2.25	3.13	0.65	1.18	1.65
1,000,000	2.32	4.26	5.90	1.21	2.23	3.09	0.63	1.16	1.61
10,000,000	2.29	4.25	6.07	1.18	2.21	3.15	0.62	1.14	1.64

Table 4. Example values of $\mathsf{R}(\mathsf{T})$ for random strings.



(a) A subset of the 200 MB "regular" files, plus English-2108.



(b) A subset of the "highly repetitive" files.

Figure 2. Distribution of LCP values (adjusted by +1 so that a log scale can be used) for two different collections of files from the Pizza & Chili Corpus. Occurrence counts are normalized relative to the file size.

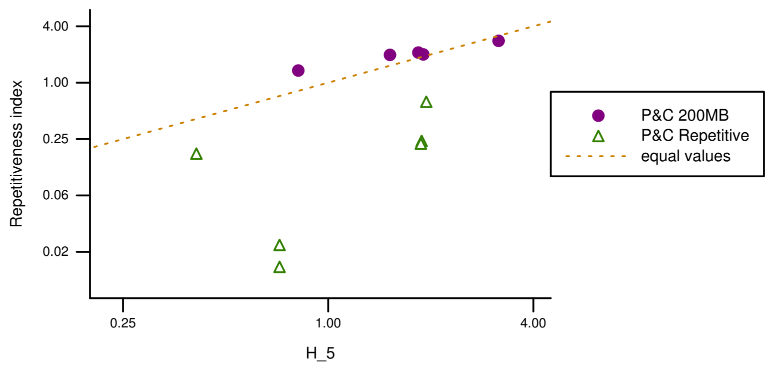


Figure 3. Correlation between H_k (with $k = 5$) and R for files in the Pizza & Chili Corpus. The H_5 values are as listed at `http://pizzachili.dcc.uchile.cl` and in Table 3.2 of `http://pizzachili.dcc.uchile.cl/repcorpus/statistics.pdf`, both accessed 20 April 2013.

Answers	$m = 8$	$m = 16$	$m = 32$	$m = 64$
Uncontrolled	$115{,}567 \pm 14{,}593$	$18{,}701 \pm 1{,}291$	$9{,}323 \pm 457$	$6{,}720 \pm 274$
$k = 1$	19.6 ± 0.6	26.3 ± 3.4	38.7 ± 1.8	61.5 ± 2.7
$k = 10$	—	53.4 ± 3.3	67.1 ± 3.2	91.1 ± 2.9
$k = 100$	—	253.9 ± 3.6	267.3 ± 3.9	292.2 ± 4.4
$k = 1{,}000$	$2{,}061.8 \pm 15.3$	$2{,}057.2 \pm 18.0$	$2{,}091.6 \pm 15.5$	$2{,}137.0 \pm 18.0$

Table 5. Time (milliseconds per set of 1,000 queries) to resolve *locate* queries against Einstein-en using an FM-INDEX, plus the standard deviation measured over 10 such pattern sets.

Table 5 shows how repetitiveness in the 446 MB file Einstein-en affects measured costs for *locate* queries. There are k and m combinations for which Einstein-en can be searched in as little as 20% of the time required to search for English-2108, shown in the bottom half of Table 3. This difference is a consequence of the internal structure of the files; in the case of English-2018, the locations in T that are returned as answers are more uniformly scattered than occurs with Einstein-en. Indeed, the long repeated strings in the Einstein-en file mean that answers form clusters, with access to each answer then being faster as a result of caching effects. The index is also smaller than the index for English-2108, and some (but not all) of the gain in speed arises from that difference, even though the asymptotic cost of querying the two structures is independent of $|\mathsf{T}|$. That is, the type of data used can also have a quite dramatic effect over measured querying costs.

Finally, note that while R is a convenient number for providing an assessment of the repetitiveness in a file, it is in no sense a lower bound on compression. Any given factor-based mechanism is free to choose the factors it uses from amongst the set of n possible factors included in the summation in Equation 4.1, and will naturally preference the longer ones, subject to the influence of the coding method used [Ferragina et al., 2009]. For example, the `xz` compression tool[6] renders all of the files depicted in Figure 3 more

[6] `http://tukaani.org/xz/`

effectively than is estimated by either H_5 or R. For the six highly repetitive Pizza & Chili Corpus files that are plotted. the ratio between R and "`xz` size, in bits per character" ranges between 1.68 and 2.31; for the five 200 MB files plotted, the ratio is between 1.11 and 1.66. Indeed, any compression tool can be used self-referentially as a measure of compressibility. The benefit of using H_k and R is that they are dependent only on the data, not the software used to process it; compared to H_k, R has the additional advantage of not needing to be accompanied by an estimate of the size of the compression model, which is the role played by σ^k.

Other approaches to defining repetitiveness are also possible. One option, for example, would be to base a definition on an LZ77-style left-to-right parse, taking into account the number of factors generated and their lengths. Kärkkäinen et al. [2013] describe an efficient process for computing LZ77 parses when the input string can be held in main memory. A second option is to compute a minimal grammar, and derive a repetitiveness score based on the complexity of the grammar. Rytter [2003] and Charikar et al. [2005] have both considered the problem of finding minimal grammars, and describe algorithms that provide approximate solutions.

5. The Importance of Scale: Massive Data Sets

Another aspect of experimentation that needs careful attention is that of scale. The memory hierarchy of a modern computer is complex, involving multiple levels with different characteristics, and pre-fetching systems that attempt to retrieve data before it is required. Comparative evaluations should be undertaken at a scale that makes use of the full spectrum of that hierarchy. For example, it is not uncommon for a modern computer to possess 16 MB or more of L3 cache; meaning that caching can play a significant part in experimental running times. Table 6 shows the measured time for *locate* queries on the English-200 test file, and illustrates this effect.

Compared to the times shown in the bottom half of Table 3, execution is around 20% faster, for the same type of source data. A similar 20–30% improvement occurs when *count* queries are measured. The difference is accounted for by the fact that the FM-INDEX for English-200 is 171 MB, whereas the FM-INDEX for English-2108 is ten times larger, at 1875 MB. Even though the cache on the test hardware is only 6 MB, it is big enough that some parts of the index that are frequently accessed remain permanently present in cache while English-200 is being searched, but to a lesser extent when English-2108 is being searched. That translates into measurably

Answers	$m = 8$	$m = 16$	$m = 32$	$m = 64$
Uncontrolled	$16{,}971 \pm 2{,}687$	771 ± 461	184 ± 198	58.3 ± 6.0
$k = 1$	18.3 ± 1.3	22.3 ± 0.6	32.2 ± 1.0	51.6 ± 2.6
$k = 10$	89.7 ± 2.3	89.4 ± 4.8	65.4 ± 2.4	71.2 ± 3.2
$k = 100$	771.4 ± 28.9	716.6 ± 20.1	—	—
$k = 1{,}000$	$7{,}136.0 \pm 190.0$	—	—	—

Table 6. Time (milliseconds per set of 1,000 queries) to resolve *locate* patterns against English-200 using an FM-INDEX, plus the standard deviation measured over 10 such pattern sets. These times can be compared with those shown in the bottom half of Table 3.

different execution times. Large data structures are also intrinsically slower to access on most modern architectures because of translation look-aside buffer (TLB) misses when memory addresses are being computed. Gog and Petri [2013] provide detailed experiments that address this issue.

To undertake valid experiments, it is thus important that the test files that are used are at a scale that fully reflects the hardware and, conversely, for referees to be sceptical of results that are demonstrated on datasets that are small relative to the hardware. For example, it would be inappropriate to test a mechanism designed for secondary storage (disk or SSD) using "just" 10 GB of data if the machine in question has 16 GB of main memory, since once the system had processed part of the query stream and had "warmed up" and filled memory, it is unlikely that any of the index would be resident on disk.

A drawback of this expectation is that generating the index for a large dataset may take substantially more resources than does using the index to process queries. The FM-INDEX falls in to this category, for example. One option experimenters should consider is to generate their indexes on one machine, and then measure execution speeds on another less well-endowed set of hardware. Another way round the dilemma is to explicitly count "disk" accesses and tabulate them as functions of *m* and *k* (see Gog et al. [2013] for an example of this approach), rather than rely solely on measured execution times.

6. Summary

As a research area, string search is relatively mature. Experimentation is regarded as being the norm, and a range of public datasets and public software implementations have been developed[7]. Based on these resources, researchers in this field are in the desirable position of being able to build on each others' work when exploring new techniques and applications.

In this article we have examined some aspects of string search experimentation, focusing in particular on the mechanism used to generate example strings, and on the way in which repetitiveness in input texts can be quantified. We demonstrated that generating test patterns by selecting random locations from within the text over-emphasizes frequently-occurring strings, and has the potential to distort experimental outcomes. The stratified approach we describe avoids this problem, and allows more precise control over experimental settings. We further demonstrated that structure within the file being searched can have a dramatic effect on measured search times, and that scale of experimentation is also an important factor that must be considered when assessing the validity of an experimental regime.

Acknowledgments: This work was supported by the Australian Research Council. Andrew Turpin participated in a number of helpful discussions.

Software and Data: The pattern generation software that has been developed during this investigation is publicly available at `http://go.unimelb.edu.au/t9en` as is the software for the FM-INDEX used in the experimentation, together with the pattern sets

[7] Some researchers argue that even this is insufficient, and that all tabulation and graphing scripts, all measured outcomes, and even the whole test environment (as a description of a virtual machine) should be captured at the time a research paper is submitted or accepted [Gent, 2013]. The page at `http://sciencecodemanifesto.org` similarly argues for a very high degree of public disclosure.

that were generated for each of the Pizza & Chili Corpus files. Details of the query timings are also available over the files in the corpus, of which we have presented only a fraction here.

References

R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In J. A. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, pages 201–210, Snowbird, Utah, 1997. IEEE Computer Society Press.

M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Information Theory*, 51(7):2554–2576, 2005.

F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. Indexes for highly repetitive document collections. In C. Macdonald, I. Ounis, and I. Ruthven, editors, *Proc. Int. Conf. Information and Knowledge Management*, pages 463–468, Glasgow, Scotland, 2011. ACM.

P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM J. Experimental Algorithmics*, 13:12:1–12:31, 2008.

P. Ferragina, I. Nitto, and R. Venturini. On the bit-complexity of Lempel-Ziv compression. In C. Mathieu, editor, *Proc. Ann. ACM-SIAM Symp. Discrete Algorithms*, pages 768–777, New York, NY, 2009. SIAM.

J. Fischer, V. Mäkinen, and N. Välimäki. Space efficient string mining under frequency constraints. In *Proc. Int. Conf. Data Mining*, pages 193–202, Pisa, Italy, 2008. IEEE Computer Society.

I. P. Gent. The recomputation manifesto. `http://arxiv.org/abs/1304.3674`, April 2013.

S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software Practice & Experience*, 2013. Preprint: `http://dx.doi.org/10.1002/spe.2198`.

S. Gog, A. Moffat, J. S. Culpepper, A. Turpin, and A. Wirth. Large-scale pattern search using reduced-space on-disk suffix arrays. *IEEE Trans. Knowledge and Data Engineering*, 2013. Preprint: `http://dx.doi.org/10.1109/TKDE.2013.129`.

J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lightweight Lempel-Ziv parsing. In V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, editors, *Proc. Int. Symp. Experimental Algorithms*, pages 139–150. LNCS 7933, Springer, 2013.

V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.*, 12(1): 40–66, 2005.

G. Navarro. Indexing highly repetitive collections. In S. Arumugam and W. F. Smyth, editors, *Proc Int. Wkshp. Combinatorial Algorithms*, pages 274–279, Tamil Nadu, India, 2012. LNCS 7643, Springer.

G. Navarro and L. Salmela. Indexing variable length substrings for exact and approximate matching. In J. Karlgren, J. Tarhio, and H. Hyyrö, editors, *Proc. Symp. String Processing and Information Retrieval*, pages 214–221, Saariselkä, Finland, 2009. LNCS 5721, Springer.

W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In A. Amir, A. Turpin, and A. Moffat, editors, *Proc. Symp. String Processing and Information Retrieval*, pages 164–175, Melbourne, Australia, 2008. LNCS 5280, Springer.