Processing of Huffman Compressed Texts with a Super-Alphabet

Kimmo Fredriksson[1]* and Jorma Tarhio[2]

[1] Department of CS, University of Joensuu
P.O. Box 111, FIN-80101 Joensuu, Finland
kfredrik@cs.joensuu.fi
[2] Department of CSE, Helsinki University of Technology
P.O. Box 5400, FIN-02015 HUT, Finland
tarhio@cs.hut.fi

Abstract. We present an efficient algorithm for scanning Huffman compressed texts. The algorithm parses the compressed text in $O(n\frac{\log_2\sigma}{b})$ time, where n is the size of the compressed text in bytes, σ is the size of the alphabet, and b is a user specified parameter. The method uses a variable size super-alphabet, with an average size of $O(\frac{b}{H\log_2\sigma})$ symbols, where H is the entropy of the text. Each super-symbol is processed in $O(1)$ time. The algorithm uses $O(2^b)$ space and $O(b2^b)$ preprocessing time. The method can be easily augmented by auxiliary functions, which can e.g. decompress the text or perform pattern matching in the compressed text. We give three example functions: decoding the text in average time $O(n\frac{\log_2\sigma}{Hw})$, where w is the number of bits in a machine word; an Aho-Corasick dictionary matching algorithm, which works in time $O(n\frac{\log_2\sigma}{b}+t)$, where t is the number of occurrences reported; and a shift-or string matching algorithm that works in time $O(n\frac{\log_2\sigma}{b}\lceil(m+s-1)/w\rceil+t)$, where m is the length of the pattern and s depends on the encoding. The Aho-Corasick algorithm uses an automaton with variable length moves, i.e. it processes variable number of states at each step. The shift-or algorithm makes variable length shifts, effectively also processing variable number of states at each step. The number of states processed in $O(1)$ time is $O(\frac{b}{H\log_2\sigma})$. The method can be applied to several other algorithms as well. We conclude with some experimental results.

1 Introduction

Huffman coding [8] is a well-known and extensively studied text compression method. It assigns for each symbol of the input a codeword, i.e. a string of bits. The more frequent the symbol is the shorter its codeword is. Huffman [8] showed how to obtain optimal *prefix codes*, i.e. no codeword is a prefix of another

* Work partially developed when the author was working in Department of Computer Science, University of Helsinki. Partially supported by the Academy of Finland.

codeword. The entropy of the alphabet Σ is

$$H = -\sum_{c \in \Sigma} p_c \log_2 p_c,$$

where p_c is the probability of a symbol c. If the input is u bytes long, then the compressed form is approximately $n = Hu$ bytes long.

In the *string matching problem*, which is common in many applications, the task is to find all the occurrences of a given string pattern in a text. Recently the *compressed matching problem* [2] has gained much attention. The trivial method is to first decompress the text and then perform the search operation. The preferred method is to search the compressed text directly, maybe taking advantage of its shorter representation. Researchers have proposed several efficient methods [3, 9, 11, 12] based on Huffman coding [8] or the Ziv-Lempel family [16, 17].

In the following we consider matching of multiple patterns in Huffman compressed texts. The work by Takeda et al. [13] gave motivation for this work. Our methods apply a super-alphabet, which means that several codewords can be processed at the same time. Fredriksson [7] got good results in speeding up standard string matching with super-alphabets. Our idea of utilizing super-alphabets in the context of Huffman compression is related with older approaches [6, 14] to speed up decompression. In particular, we apply super-alphabets to the Aho-Corasick algorithm [1] and to the shift-or algorithm [4, 15].

The Aho-Corasick automaton (the AC automaton for short) is extended to make transitions with compressed character tuples, the size of the tuple depending on the entropy H. We also extend the shift-or string matching algorithm to make variable length shifts, again the shift length depending on H. Hence the better the compression ratio is the faster the algorithms run. We first give an efficient algorithm for scanning Huffman compressed texts. The method can then be easily augmented by auxiliary functions, which can e.g. decompress the text or perform pattern matching in the compressed text.

All the algorithms are linear in u and n, where u and n are the lengths (in bytes) of the original and compressed texts, respectively, i.e. they are $O(n) = O(u)$, but they can be said to be sublinear in the sense that the number of steps is less than n or u by a constant factor. For example, the AC automaton runs in $O(uH \log_2 \sigma / b) = O(n \log_2 \sigma / b)$ steps, where b is a user chosen (constant) parameter. We use this notion of sublinearity.

2 Preliminaries

Let a *text* $W[1..u]$ be a string in an alphabet Σ of size σ. The Huffman compressed text of W is denoted by $T[1..n]$. The number of characters in W and the number of codewords in T is u. The compressed text T occupies only $n < u$ bytes of storage. Similarly, a *pattern* $P[1..m]$ is a string in the same alphabet Σ. A *dictionary* $\mathcal{D} = \{P_1, P_2, ..., P_d\}$ is a set of d patterns. For simplicity and without loss of generality, we assume that each pattern in $\mathcal{D}$ has the same length m.

The alphabet Σ may contain symbols that do not appear in W. This is a typical situation, consider, e.g. a natural language text in the ASCII alphabet. The codewords are generated only for those symbols that appear in W. At this point one should note that if the pattern contains symbols that have no codeword, then the pattern cannot have any exact occurrences in W.

3 Basic tools

Before describing the actual algorithms, we introduce a method for scanning the compressed text efficiently. The algorithms will process and "decode" the compressed input in blocks of b bits. Our approach is more general than that of [14], where only $b = 8$ is considered. Let us denote a block of b bits by Q. The block Q may encode several, one, or no original symbols of the original text, depending on b and on the length of the codewords. In the best case, Q encodes b symbols, each having a codeword of length 1. In addition, Q may contain a prefix of another codeword. We denote the ith bit of Q by Q_i.

We associate with Q several attributes, which are preprocessed and stored for every possible Q.

- $s(Q)$: the bit-pattern Q encodes $s(Q)$ symbols, $0 \leq s(Q) \leq b$.
- $r(Q)$: the number of remaining bits in Q, i.e. the length of the proper prefix of the next codeword; Q contains this prefix.
- $h(Q)$ the node in the Huffman tree after traversing the tree with the prefix $Q_{b-r(Q)+1} \cdots Q_b$ of length $r(Q)$.

Q may have some additional attributes, depending on the algorithm. For example, if we wanted to decode the original text, we could add an attribute $c(Q)$ that stores the $s(Q)$ symbols of Σ that Q encodes.

Having this information for every Q, it is easy and efficient to traverse through the compressed file, keeping track of the codeword boundaries. Q is initialized with the first b bits of the compressed input string. Then if $s > 0$, we do whatever processing is required (e.g. dump $c(Q)$), shift the bits in Q to the right by $b-r(Q)$ bit positions, and pad Q with the next $b - r(Q)$ bits.

If $s = 0$, i.e. the whole Q was a prefix of a codeword, then we jump to the node $h(Q)$ and continue the "decoding" from there, until we reach the leaf (and e.g. dump the symbol stored in the leaf). Then we continue without the tree.

Our scanning algorithm is somewhat similar to the one given by Choueka et al. [6], but ours is simpler. Particularly, they process the remaining bits in a different way. They construct a separate look-up table of Q for each possible number of remaining bits of the previous step. Thus their approach consumes more memory than ours.

The basic scanning algorithm is given in pseudocode in Alg. 1. For simplicity of presentation, the algorithm implicitly assumes that $b = 8$, i.e. Q contains one byte, but it is trivial to modify the algorithm to use general b, while maintaining constant time updates of Q.

We use the following notation. A machine word has w bits, numbered from the least significant bit to the most significant bit. For bit-wise operations of words a C-like notation is used, & is `and`, | is `or`, and << and >> are shift to left and shift to right with zero padding.

The scanning algorithm takes an auxiliary function $A(Q, h(Q))$ as a parameter and this function is called whenever at least one symbol has been "decoded". We do not describe the actual processing done in A, but only parameter passing. If the first parameter Q is not zero, then Q denotes the current bit-pattern the scanning algorithm has read and it encodes ≥ 1 symbols of the original text. Otherwise, Q is a prefix of a codeword and the algorithm traverses into a leaf of the Huffman tree bit by bit so that $h(Q)$ is a pointer to that leaf, which holds one original symbol. Otherwise we pass $h(Q) = \textsc{null}$. With this function schema it is easy to implement other algorithms that process the compressed data. For example, the decoding function is given in Alg. 2.

The time complexity of the scanning algorithm is

$$O\left(u\frac{H\log_2\sigma}{b}\right) = O\left(n\frac{\log_2\sigma}{b}\right),$$

as there are u characters in the original text, one codeword takes $O(H\log_2\sigma)$ bits, where H is the entropy, and we process $O(b)$ bits at a single step.

Alg. 2 can output $\lfloor w/\log_2\sigma\rfloor$ characters at a time. There are $O(b/(H\log_2\sigma))$ symbols to output on average. Hence, the complexity of the decoding algorithm is

$$O\left(n\frac{\log_2\sigma}{b}\frac{\log_2\sigma}{w}\frac{b}{H\log_2\sigma}\right) = O\left(n\frac{\log_2\sigma}{Hw}\right).$$

Note that this analysis is valid only for b large enough, i.e. $b/(H\log_2\sigma) > w/\log_2\sigma$ should hold, i.e. $b > Hw$. Otherwise it is $O(u)$ in the worst case.

In the sequel we give functions A that perform efficient (dictionary) pattern matching in the compressed text.

Preprocessing the attributes. The attributes can be computed by generating every possible Q, in time $O(2^b)$, and then traversing the Huffman tree in time b for each generated Q. Every time we arrive into a leaf, we update the (basic) attribute information in time $O(1)$ (for some other attributes, depending on the application, the updating can take more time). Finally, $h(Q)$ is the node where we ended up after b steps. Hence, the total time is $O(b2^b)$, which is affordable for small values of b. In practice the optimal b is between 8–14, see Sec. 6. In our implementation in C, we have an array of `struct`s for the attributes, indexed with Q.

4 Aho-Corasick automaton

The AC automaton [1] is a finite automaton for searching a set of pattern from a text of length u in time $O(u + t)$, where t is the number of occurrences found.

Alg. 1 Scan($A(Q, v)$).

Input: auxiliary function $A(Q, v)$
Output: output of A

```
$i \leftarrow 1$, $Q \leftarrow T[1]$, $l \leftarrow b$, $v \leftarrow$ NULL
while $i \leq n$                          // ANY INPUT LEFT?
    if $l < b$                            // LESS THAN $b$ BITS LEFT IN $Q$?
        $i \leftarrow i + 1$
        if $i \leq n$
            $Q \leftarrow Q \mid (T[i] << l)$ // APPEND $Q$ WITH THE NEXT $b$ BITS
            $l \leftarrow l + b$
    $Q' \leftarrow Q \,\&\, ((1 << b) - 1)$     // PUT THE $b$ FIRST BITS OF $Q$ TO $Q'$
    if $v =$ NULL                         // TRAVERSING THE HUFFMAN TREE?
        if $s(Q') > 0$                    // DOES $Q'$ CODE ANY SYMBOL?
            $A(Q', \text{NULL})$              // YES, CALL THE AUXILIARY FUNCTION
            $s \leftarrow b - r(Q')$          // $s$ OUT OF $b$ BITS PROCESSED
        else                              // $Q'$ WAS JUST A PREFIX, SO...
            $v \leftarrow h(Q')$              // JUMP TO NODE $v$ IN THE TREE
            $s \leftarrow b$                  // $b$ BITS PROCESSED
        $Q \leftarrow Q >> s$                 // REMOVE THE PROCESSED BITS
        $l \leftarrow l - s$
    else                                  // TRAVERSING THE HUFFMAN TREE
        if $v$ is a leaf
            $A(0, v)$
            $v \leftarrow$ NULL               // CONTINUE W/O THE TREE
        else
            if $Q \,\&\, 1 = 0$
                $v \leftarrow left(v)$
            else
                $v \leftarrow right(v)$
        $Q \leftarrow Q >> 1$                 // REMOVE THE PROCESSED BIT
        $l \leftarrow l - 1$
```

Alg. 2 Dump(Q, v).

Input: bit-pattern Q, leaf v of the Huffman tree
Output: the symbols coded by Q or v

```
if $v =$ NULL                             // USING THE HUFFMAN TREE?
    output($c(Q)$)                        // NO, OUTPUT THE SYMBOLS OF $Q$
else                                      // $v$ IS A LEAF,
    output($v.symbol$)                    // SO OUTPUT THE SYMBOL
```

The automaton contains $O(dm)$ states. It can be made deterministic, so that each node has a transition for each character in the alphabet. We build directly a deterministic automaton in time $O(dm\sigma)$, and the result uses $O(dm\sigma)$ space.

Our algorithm does effectively the same as the algorithm in [13], but is simpler and more efficient.

Transitions with compressed character tuples. Each node node v in the automaton has transition $\delta(v, c) \to v'$, for each $c \in \Sigma$. In [7] a super-alphabet method was used to simulate deterministic automata efficiently. The basic idea is that when a *set* of characters is interpreted as one super-symbol, i.e. a catenation of the original symbols, it is possible to precompute a *shortcut* transition to the state where the original transitions would lead with that set of the original symbols. In our case, Q implicitly codes a varying length super-symbol, the length for a specific Q being $s(Q)$. We therefore precompute the shortcut transitions $\Delta(v, Q)$ for each state v of the automaton and for each possible Q. It is also possible that this shortcut transition goes through several accepting states of the original automaton. To take this into account, we collect the output information into an attribute $o(v, Q)$. The preprocessing time for a straight-forward method is $O(dm(\sigma + \frac{b}{H \log_2 \sigma} 2^b) + b2^b)$, which is reasonable for small b. This can be improved to $O(dm(\sigma + 2^b) + b2^b)$ by using incremental evaluation of the shortcut transitions.

Alg. 3 gives the auxiliary function for the AC automaton. The total time of the algorithm is $O(t)$, where t is the number of matches reported, or $O(1)$, if we only count the number of matches. Hence the expected time of Alg. 1 with Alg. 3 is

$$O\left(n \frac{\log_2 \sigma}{b} + t\right),$$

which is sublinear for most reasonable choices of parameters.

Alg. 3 AC(Q, v).

Input: bit-pattern Q, leaf v of the Huffman tree
Output: matches, if any

```
1   if $v$ = NULL                             // USING THE HUFFMAN TREE?
2       $state' \leftarrow \Delta(state, Q)$              // NO, MAKE SUPER-TRANSITION
3       if $o(state, Q)$ has outputs
4           output the occurrences
5   else                                      // $v$ IS A LEAF,
6       $state' \leftarrow \delta(state, v.symbol)$       // SO MAKE A NORMAL TRANSITION
7       if $o(state, v.symbol)$ has outputs
8           output occurrences
9   $state \leftarrow state'$                         // UPDATE THE CURRENT $state$
```

Takeda et al. [13, 14] use a separate DFA to distinguish the beginning of codewords. We do not need such a DFA, because the codewords are synchronized on line.

5 Shift-or

Shift-or is a well-known bit-parallel string matching algorithm [4, 15]. In [7] a super-alphabet simulation method for shift-or algorithm was given. Here we briefly review the normal shift-or algorithm and then show how we can use Q and variable length shifts to directly search the pattern in Huffman compressed data.

The algorithm is based on a non-deterministic finite state automaton, which is constructed as follows. The automaton has states $1, 2, \ldots, m+1$. The state 1 is the initial state, state $m+1$ is the final (accepting) state, and for $i = 1, \ldots, m$ there is a transition from the state i to the state $i+1$ for character $P[i]$. In addition, there is a transition for every $c \in \Sigma$ from and to the initial state.

The preprocessing algorithm builds a table E. The table has one bit-mask entry for each character in the alphabet. For $1 \leq i \leq m$, the mask $E(c)$ has the ith bit set to 0, if and only if $P[i] = c$, and to 1 otherwise. The bit-mask table corresponds to the transitions of the implicit automaton. That is, if the bit i in $E(c)$ is 0, then there is a transition from the state i to the state $i+1$ with character c.

We also need a bit-vector D for the states of the automaton. The ith bit of the state vector is set to 0, if and only if the state i is active. Initially each bit in the state vector is set to 1. For each new text symbol c, we update the vector as follows:

$$D \leftarrow (D << 1) \mid E(c)$$

Each state gets its value from the previous one $(D << 1)$, which remains active only if the text character matches the corresponding transition $(\mid E(c))$. The first state is set active automatically by the shift operation, which sets the least significant bit to 0. If after the simulation step, the mth bit of D is zero, then there is an occurrence of P.

Clearly each step of the automaton is simulated in time $O(\lceil m/w \rceil)$, which leads to $O(n\lceil m/w \rceil)$ total time.

5.1 Super-alphabet and variable length shifts

It is possible to improve the NFA simulation by using bit-parallelism also in accessing the table E. In effect we use a variable length super-alphabet of size $\sigma^{s(Q)}$. It should be emphasized that we do *not* modify the automaton, that is, the automaton does *not* use any super-alphabet, but the super-alphabet is only used to simulate the original automaton faster.

Let $c_1, \ldots, c_{s(Q)}$ be the original symbols encoded by a block Q. We set

$$\begin{aligned} B(Q) \leftarrow & ((E(c_1) \ \& \ 1^m) << (s(Q)-1)) \\ & \mid \ ((E(c_2) \ \& \ 1^m) << (s(Q)-2)) \mid \ \ldots \\ & \mid \ (E(c_{s(Q)}) \ \& \ 1^m). \end{aligned}$$

In other words we simulate what happens in $s(Q)$ steps, which we bit-wise `or` into the vector D of the basic algorithm. The simulation step is

$$D \leftarrow (D << s(Q)) \mid B(Q).$$

Now $B(Q)$ pre-shifts and bit-wise ors the state transition information for $s(Q)$ consecutive original symbols and the state vector D is then updated with this precomputation.

After the simulation step, any of the bits numbered $m, \ldots, m + s(Q) - 1$ may be zero in D. This indicates a pattern occurrence with an occurrence location offset of $s(Q) - 1, \ldots, 0$ corresponding to the offset of the zero bit in D.

The table B is easy to compute in time $O(b2^b)$. The bit-vectors have the length $m + s - 1$, where $s = \max_Q(s(Q))$, due to the need of $s - 1$ extra bits to handle the super-alphabet. Alg. 4 gives the auxiliary function for shift-or with variable length shifts.

Alg. 4 Shift-or(Q, v).

Input: bit-pattern Q, leaf v of the Huffman tree
Output: matches, if any

```
if $v$ = NULL                              // USING THE HUFFMAN TREE?
    $D \leftarrow D << s(Q) \mid B(Q)$     // NO, MAKE A SUPER-SHIFT
    $msk \leftarrow (1 << (m + s(Q)) - 1) - (1 << (m - 2))$
    if $D$ & $msk \neq msk$                // ANY BIT $m, \ldots, m + s(Q) - 1$ ZERO?
        check the bits individually and output matches...
else                                       // $v$ IS A LEAF,
    $D \leftarrow D << 1 \mid B(v.symbol)$ // SO MAKE A NORMAL SHIFT
    $msk \leftarrow 1 << (m - 1)$
    if $D$ & $msk \neq msk$                // IS BIT $m$ ZERO?
        output match...
```

Alg. 4 runs in time $O(\lceil (m + s - 1)/w \rceil + t)$, where t is the number of matches reported, m is the length of the pattern, and w is the length of the machine word in bits (typically 32 or 64), or in time $O(\lceil (m + s - 1)/w \rceil)$, if we only count the number of matches. This is $O(1)$ for short patterns. Hence the expected time of Alg. 1 with Alg. 4 is

$$O\left(n \frac{\log_2 \sigma}{b} \lceil (m + s - 1)/w \rceil + t\right),$$

which is sublinear for most reasonable choice of parameters.

5.2 Multiple patterns

In [5, 10] the problem of searching multiple patterns is reduced to searching a single "super-imposed" pattern, using classes of characters. This approach is fairly straightforward to combine with the super-alphabet shift-or. The basic idea is that each text symbol is allowed to match to the symbol of position j in *any* of the patterns. For example, if we have patterns "Hello" and "world", then we form a super-pattern "{H,w}{e,o}{l,r}{l}{o,d}". This matches for example text strings "wello", "Horlo", and "Helld". This is obviously a filter and thus the potential matches require verification. The bright side of this is that the algorithm is very simple; in fact it does not change at all. The only necessary thing is to alter the preprocessing. After we have computed the E_i table for each pattern $P_i \in \mathcal{D}$, we just take the union of the zero bits:

$$E \leftarrow E_1 \ \& \ E_2 \ \& \ \ldots \ \& \ E_d.$$

The E table can then in turn be used to produce the B table.

Assuming uniform, independent distribution of characters, the probability that the super-pattern matches, is

$$\left(1 - \left(1 - \frac{1}{\sigma}\right)^d\right)^m < \left(\frac{d}{\sigma}\right)^m,$$

where the inequality holds for $d < \sigma$. However, note that the assumption of uniform distribution of characters in fact contradicts with our assumption that the text is compressible. We therefore replace σ in the analysis with $1/p$, where p is the probability that two symbols match.

The filter performs well, if the number of patterns is not too large and the patterns are reasonably long. For too long patterns ($m > w - s + 1$) we can use the suffixes of length $w - s + 1$.

As the multi-pattern shift-or is a filter, the occurrences have to be verified. For that we compress the patterns and build a binary trie of the reverse of the compressed patterns. The verification is then just a matter of scanning the compressed text backwards from the bit that triggered the verification, matching the bits against the trie until a match is found, or some bit mismatches. This requires $O(m \log_2 \sigma)$ worst case time or

$$O\left(\log_2 d + \sum_{i=1}^{m \log_2 \sigma - \log_2 d} \left(\frac{1}{2}\right)^i\right) = O(\log_2 d)$$

average time. This holds as far as $\log_2 d < m \log_2 \sigma$, or equivalently, $d < \sigma^m$. Otherwise we replace $O(\log_2 d)$ with $O(m \log_2 \sigma)$. However, at this point the filter would not work anymore. This can be improved to $O(\log_2 d/b)$ average time by using a super-alphabet. The total verification cost on average is therefore

$$\frac{u(1 - (1-p)^d)^m \log_2 d}{b} < \frac{u(pd)^m \log_2 d}{b},$$

where p is the probability that two symbols match, and the inequality holds for $d < 1/p$.

5.3 Applying q-grams to shift-or

To overcome the problem of decreased filtering capability for large number of patterns (and especially with combination of small alphabets) of the previous method, we use the method suggested in [10]. That is, we apply the super-alphabet approach in a different way. We take all the consecutive non-overlapping q-grams (substrings of length q) from the pattern(s) and build the shift-or automaton using these 'super-characters'. (Note that overlapping q-grams were used in [10].)

Consider a pattern $P =$"qwerty" and $q = 2$. A straightforward implementation of this idea would lead to synchronization problem, only every qth position of the text would be inspected, and some matches would be lost. This automaton is illustrated in Fig. 1.

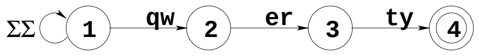


Fig. 1. Non-synchronized 2-gram automaton for $P =$"qwerty".

We fix this problem by generating q versions, i.e. different alignments, of the pattern(s). The pattern is shifted q times to the right, one character position at a time, and for each "frame-shift", we obtain a new version of the pattern. The shifting operation introduces wild card symbols '?', which match for every character. We have to also pad each generated pattern to the length $\lceil m/q \rceil q$ with '?' symbols. For example, if $P =$"qwerty" and $q = 2$, we obtain patterns "qwerty" and "?qwerty".

We can search all the patterns at the same time using the shift-or algorithm [4]. All the patterns can be packed in a single computer word with $(m/q)q + q - 1 = m + q - 1$ bits. The resulting automaton is illustrated in Fig. 2.

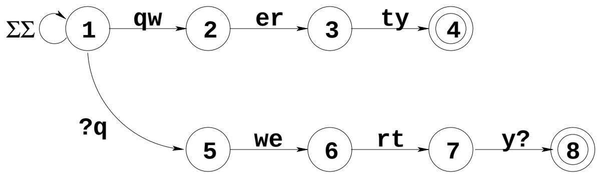


Fig. 2. Synchronized 2-gram automaton for $P =$"qwerty".

The automaton can be simulated in $O(u\lceil (m + q - 1)/w \rceil /q)$ time for uncompressed text, which leads to

$$O\left(n\frac{\log_2 \sigma}{b}\left\lceil\left(m + \left\lceil\frac{b}{H\log_2 \sigma}\right\rceil - 1\right)/w\right\rceil + t\right)$$

time for compressed search, if we choose $q = \Theta(b/(H \log_2 \sigma))$. The (implementation) problem with this approach is that the shifts cannot be variable length now, so the shift must be delayed until at least q symbols are decoded.

Assuming again that the probability that two symbols match is p, then the probability that the q-gram super-pattern matches is now only at most

$$(1 - (1 - p^q)^{qd})^{\lfloor m/q \rfloor - 1} < (qdp^q)^{\lfloor m/q \rfloor - 1},$$

where the inequality now holds for $qd < 1/p^q$. That is, the probability that a super-symbols matches is p^q, there are qd patterns in total and the number of super-symbols without the wild card '?' is at least $\lfloor m/q \rfloor - 1$ for each pattern. This allows much larger pattern sets, especially for small alphabets.

6 Experimental result

We use the files `bible.txt`, `E.coli`, and `world192.txt` in the large corpus, available in `http://corpus.canterbury.ac.nz/descriptions/`. The shortest file is `world`, which is 2473400 bytes. In able to compare the search speeds better, we truncated all three files to this length before compressing. The compression ratios (the compressed size per the original size) of the files are: `world`: 63.0%; `bible`: 54.8%; `ecoli`: 25.0%.

We have implemented the algorithms in C, compiled using `gcc 3.1` with full optimizations. The experiments were run in 2GHz Pentium 4, with 512MB RAM, with Linux 2.4. The q-gram version of the shift-or algorithm (Sec. 5.3) has not yet been implemented. The timings are averages over 10 runs. For AC and the multi-pattern shift-or, we randomly picked 10 or 1000 patterns and for the plain shift-or one pattern. The pattern lengths were 8 characters or more for `bible` and `world` and 16 characters for `ecoli`.

time (s)	ecoli	bible	world
shift-or	0.10	0.10	0.10
AC	0.04	0.03	0.03

Table 1. Execution times with uncompressed texts.

Fig. 3 shows the search times for the shift-or algorithm and AC automaton, for varying b. As expected, the time quickly decreases with increasing b, up to a certain point. When b gets too large, the cache effects become important; the preprocessed attributes do not fit in the cache anymore and the time begins to increase. This effect is more emphasized in the AC case, as the transition function consumes a lot of memory for large b. The results show also the predicted fact that the better the compression ratio is the faster the searching is. The preprocessing times are practically negligible (< 0.01s for $b \leq 16$) for shift-or

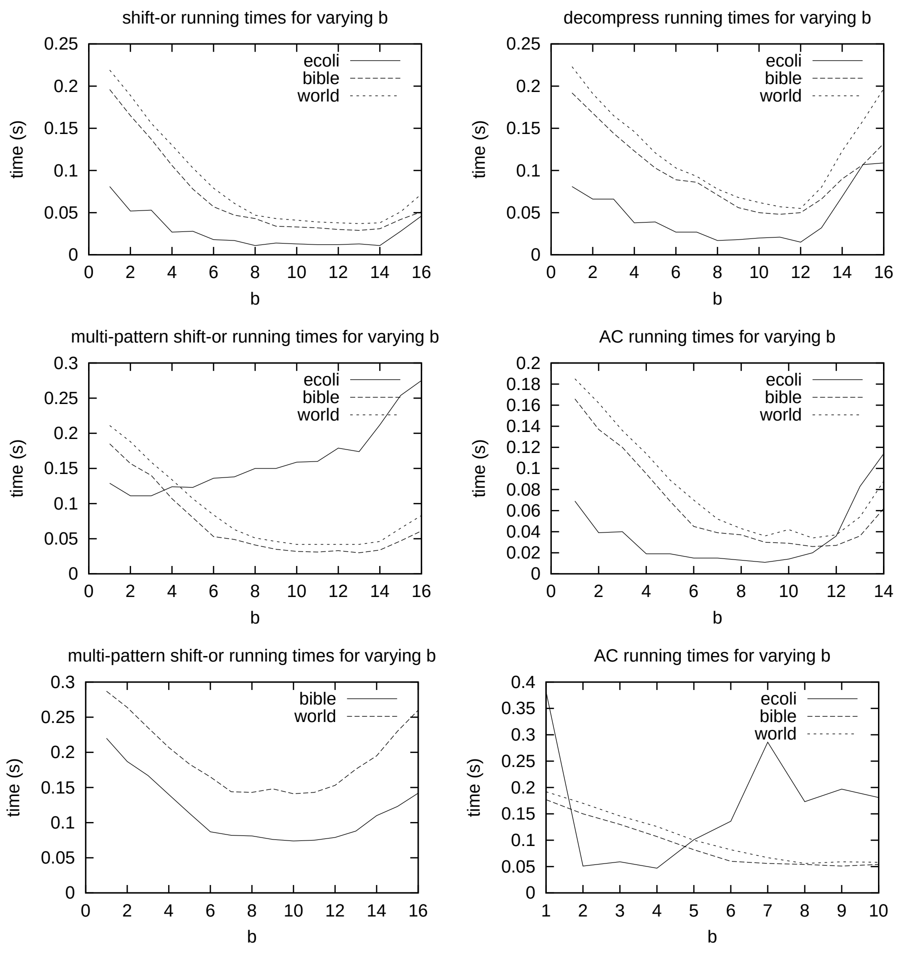
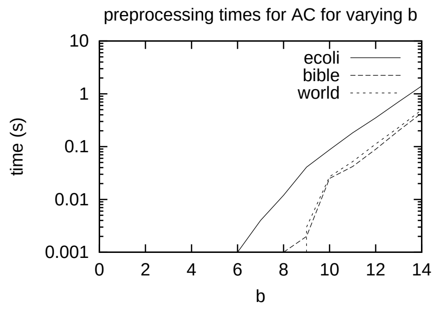


Fig. 3. Execution times in seconds for varying b. From top to bottom, left to right: shift-or (single pattern), decompression, shift-or (10 patterns), AC (10 patterns), shift-or (1000 patterns), AC (1000 patterns).

and decompression algorithms, for AC the times rapidly grow after $b > 8$, but fortunately the AC running times for $b = 8$ are almost optimized.

For comparison, we run the basic version of the shift-or and the AC algorithm for uncompressed texts. The results are shown in Table 1. The algorithms only count the number of matches, without reporting them. Our AC automaton is a simple and optimized deterministic version, containing only two lines in a `for`-loop. The searching times are comparable, but when adding the decompression times, the direct matching in compressed texts becomes much faster. Our shift-or algorithm for compressed texts is extremely fast, much faster than for uncompressed texts, with the sole exception of the multi-pattern version with



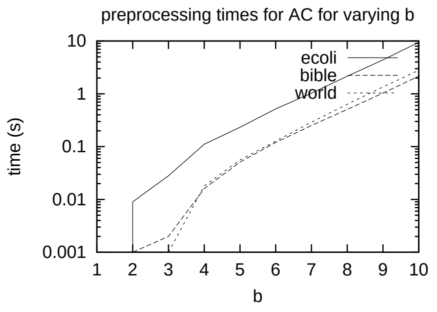


Fig. 4. AC preprocessing times in seconds for varying b. Left: for 10 patterns. Right: for 1000 patterns.

DNA alphabets. Even this should become competitive if the q-gram version were used.

7 Conclusions

We have presented an efficient algorithm for scanning Huffman compressed texts. The method can be augmented with many existing algorithms for non-compressed texts, resulting in fast algorithms for compressed texts. In particular, we have obtained fast string matching algorithms. The algorithms are novel in that they process variable length character tuples in constant time. The average length of the tuples depends on the entropy and hence the better the compression ratio is the faster the algorithms are. Our experiments show that an adjustable b almost always leads to faster solutions than the fixed $b = 8$ used earlier.

One problem of Huffman coding is that the compression ratio is poor on natural language texts. However, this problem can be downgraded by using the *words* of a text as an alphabet. This was utilized by Moura et al. [11]. Instead of using binary Huffman codes, they use *tagged* byte-oriented codes to speed up decompression and searching. One bit of each byte was used as a tag to mark the the first byte of a codeword. This slightly increases the size of the compressed files but allows searching of Boyer-Moore type. Using words as the alphabet and standard binary codes (instead of byte codes), together with our approach, we can achieve even higher compression ratios with reasonably high-speed decompression and searching.

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.
2. A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proceedings of 2nd IEEE Data Compression Conference (DCC'92)*, pages 279–288. IEEE Computer Society Press, 1992.

3. A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *J. Comput. Syst. Sci.*, 52(2):299–307, 1996.
4. R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
5. R. A. Baeza-Yates and G. Navarro. Multiple approximate string matching. In F. K. H. A. Dehne, A. Rau-Chaplin, J.-R. Sack, and R. Tamassia, editors, *Proceedings of the 5th Workshop on Algorithms and Data Structures*, number 1272 in Lecture Notes in Computer Science, pages 174–184, Halifax, Nova Scotia, Canada, 1997. Springer-Verlag, Berlin.
6. Y. Choueka, S.T. Klein, and Y. Perl. Efficient variants of Huffman codes in high-level languages. In *Proceedings of SIGIR'85, 8th Annual International Conference of Research and Development in Information Retrieval*, pages 122–130. ACM, 1985.
7. K. Fredriksson. Faster string matching with super-alphabets. In *Proceedings of String Processing and Information Retrieval (SPIRE'02)*, Lecture Notes in Computer Science 2476, pages 44–57. Springer-Verlag, 2002.
8. D. A. Huffman. A method for the construction of minimum redundancy codes. *Proc. I.R.E.*, 40:1098–1101, 1951.
9. S.T. Klein and D. Shapira. Pattern matching in Huffman encoded texts. In *Proceedings of 11th IEEE Data Compression Conference (DCC'01)*, pages 449–458. IEEE Computer Society Press, 2001.
10. J. Kytöjoki, L. Salmela, and J. Tarhio. Tuning string matching for huge pattern sets. In *Proceedings of Combinatorial Pattern Matching (CPM'03)*, Lecture Notes in Computer Science 2676, pages 211–224. Springer-Verlag, 2003.
11. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.
12. G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. In *Proceedings of 11th IEEE Data Compression Conference (DCC'01)*, pages 459–468. IEEE Computer Society Press, 2001.
13. M. Takeda, S. Miyamoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa. Processing text files as is: Pattern matching over compressed texts, multi-byte character texts, and semi-structured texts. In *Proceedings of String Processing and Information Retrieval (SPIRE'02)*, Lecture Notes in Computer Science 2476, pages 170–186. Springer-Verlag, 2002.
14. M. Takeda, Y. Shibata, T. Matsumoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa. Speeding up string pattern matching by text compression: The dawn of a new era. *Transactions of Information Processing Society of Japan*, 42(3):370–384, 2001.
15. S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.
16. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.
17. J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inf. Theory*, 24:530–536, 1978.