Let Sleeping Files Lie: Pattern Matching in Z-compressed Files

Amihood Amir* Georgia Tech

Gary Benson† U. of Southern California

Martin Farach‡ DIMACS and Rutgers

July 13, 1993

Abstract

The current explosion of stored information necessitates a new model of pattern matching, that of *compressed matching*. In this model one tries to find all occurrences of a pattern in a compressed text in time proportional to the compressed text size, *i.e.*, without decompressing the text.

The most effective general purpose compression algorithms are *adaptive*, in that the text represented by each compression symbol is determined dynamically by the data. As a result, the encoding of a substring depends on its location. Thus the same substring may "look different" every time it appears in the compressed text.

In this paper we consider pattern matching without decompression in the UNIX Z-compression. This is a variant of the Lempel-Ziv adaptive compression scheme. If n is the length of the *compressed* text and m is the length of the pattern, our algorithms find the first pattern occurrence in time $O(n + m^2)$ or $O(n \log m + m)$.

We also introduce a new criterion to measure compressed matching algorithms, that of *extra space*. We show how to modify our algorithms to achieve a tradeoff between the amount of extra space used and the algorithm's time complexity.

*College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280; (404) 853-0083; amir@cc.gatech.edu; Partially supported by NSF grant CCR-92-23699 and IRI-90-13055.

†Department of Mathematics, University of Southern California, DRB 155, 1024 W. 36th Pl., Los Angeles, CA 90089-1113; (213) 740-2404; gbenson@hto.usc.edu; Partially supported by NSF grant DMS-90-05833.

‡DIMACS, Box 1179, Rutgers University, Piscataway, NJ 08855; (908) 932-5928; farach@dimacs.rutgers.edu; Supported by DIMACS under NSF contract STC-88-09648.

1 Introduction

Recent years have seen an explosion in the gathering and storage of data. Some institutions now have data archives that are too vast to be processed. Because of the prodigous amounts of data, virtually all of it is maintained in some compressed form.

This proliferation of stored information introduces a *new demand* on data compression schemes. It is no longer sufficient that the compression achieve a good ratio and that the compression algorithm be fast. We now need algorithms for **pattern matching in time and space proportional to the compressed size**, i.e. without the need to decompress.

The *compressed matching problem* was formally defined by Amir and Benson [2, 1] as follows: Let $\sigma = s_1 \cdots s_u$ be a *text string* of length u over alphabet $\Sigma = \{a_1, \ldots, a_q\}$. Let $\sigma.c = t_1 \cdots t_n$ be a *compression* of σ of length $n \leq u$.

INPUT: Compressed text $\sigma.c = t_1 \cdots t_n$, and pattern $P = p_1 \cdots p_m$.

OUTPUT: The first text location i such that there is a pattern occurrence at s_i, i.e. $s_{i+j-1} = p_j, \quad j = 1, \ldots, m$.

Amir and Benson [2, 1] also defined a compressed matching to be *efficient* if its time complexity is $o(u)$, *almost optimal* if its time complexity is $O(n \log m + m)$ and *optimal* if it runs in time $O(n+m)$.

The first compressed matching algorithms were side effects of papers by Eilam-Tsoreff and Vishkin [6] and Amir, Landau and Vishkin [4]. The techniques of Eilam-Tsoreff and Vishkin give a trivial optimal algorithm for compressed *string* matching under the run-length compression. Amir, Landau and Vishkin showed an efficient algorithm for *two dimensional* compressed matching. That algorithm's running time was $o(u)$ but $\Omega(\frac{u}{m})$. Since the two dimensional run length compression size may be as small as $n = \sqrt{u}$, their algorithm is clearly far from optimal. Amir and Benson [2, 1] developed an almost optimal $O(n \log m)$ algorithm for two dimensional run-length compression, and Amir, Benson and Farach [3] devised an optimal $O(n)$ algorithm for this problem.

The main difficulty in the latter algorithms is due to the fact that the text is two dimensional. In comparison, the algorithm for compressed *string* matching is straight-forward. In fact, any *non-adaptive* one dimensional compression, such as run-length or Huffman encoding, has an easy compressed matching algorithm. Simply compress the *pattern* and run your favorite string matching algorithm on the compressed text and pattern. There may be a need for some extra work. For example, the first and last pattern elements have to be handled separately in the run-length compression case, and the starting bit of each encoded symbol needs to be found in the Huffman encoding.

The challenge seems to be *adaptive compressions* such as Lempel-Ziv [12]. In an adaptive compression, the text represented by each compression symbol is determined dynamically by the data. As a result, the same substring will be encoded differently depending on its location in the text. Thus encoding the pattern is futile since it will not appear in the compressed text in its compressed form.

Algorithm	*Section*	*Time*	*Extra Space*
Currently known[1]		$O(u)$	$O(n+m)$
Optimal for $m \leq \sqrt{n}$	4.2	$O(n+m^2)$	$O(n+m^2)$
Almost Optimal	4.3	$O(n \log m + m)$	$O(n+m)$
Hashing implementation, $c < m, \sqrt{u}$	5.2	$O(nc^2+m^2)$ w.h.p.	$O(\frac{n}{c}+m^2)$
Hashing implementation for text files	6	$O(n \log^2 n + m^2)$ expected	$O(m^2)$
Worst case tradeoff	5.2	$O(n\alpha\beta + n\beta \log m + m^2)$ $\alpha = \min(c, m),\ \beta = \min(c, \sqrt{u})$	$O(\frac{n}{c}+m^2)$

Table 1: Summary of Results

In this paper we consider one dimensional compressed matching in the UNIX Z-compression (also known as the LZW compression). This is a variation of the Lempel-Ziv compression introduced by Welch [11]. It is also an adaptive compression but encoding and decoding is much simpler than in the Lempel-Ziv compression.

The **main contributions** of our paper are:

- We show the first known **almost optimal** compressed matching algorithm for an **adaptive compression**. The algorithm is based on some new insights into the nature of the LZW compression. Our algorithm is simple, fast and easy to code. Our constants are **smaller** than those in the naive "decompress-then-search" option.

- We introduce a **new criterion** for evaluating compressed matching algorithms, that of *extra space*. In some applications there is a limited amount of extra available space. Therefore, an optimal algorithm that uses $O(n)$ space, *in addition to the compressed file itself*, may not be feasible. We need algorithms that use $o(n)$ extra space, optimally $O(m)$, in addition to the n-length compressed file.

- We show how to modify our basic algorithm to achieve **tradeoffs between time and the amount of extra space used**. Our results are summarized in table 1.

The paper is constructed as follows. Section 2 defines the LZW compression and proves some interesting properties of that compression. Section 3 is the outline of our basic algorithm. In section 4 we discuss some implementations that yield almost optimal algorithms. Section 5 introduces the extra space criterion and analyses some time/extra space tradeoffs. We discuss some "real-life" issues in section 6 and conclude with a plethora of open problems in section 7.

[1] This is essentially the algorithm that decompresses and searches in the decompressed string. The reason the additional space is $O(n+m)$ rather than $O(u+m)$ is that it is not necessary to keep the entire decompressed text. It is sufficient to store a sliding window of m symbols. Such an algorithm is implemented in unix by the command `zcat $file | grep $pattern`.

2 Preliminaries

Definitions: Let $\sigma = s_1 \cdots s_u$ be a string over alphabet $\Sigma = \{a_1, \ldots, a_q\}$. The *Z-compression* $\sigma.Z$ *of* σ is the string $\sigma.Z[1] \cdots \sigma.Z[n]$, where $1 \leq \sigma.Z[i] \leq n+q-1$, for $i = 1, \ldots, n$. The Z-compression is constructed with the aid of a *dictionary trie* T_σ. The dictionary is an $(n+q+1)$-node tree whose nodes are *labeled* by symbols of Σ, and *numbered* 0 through $n+q$. The *string of a node* p in T_σ is the concatenation of the node labels on the path from the root to p. The dictionary and compression string are constructed as follows:

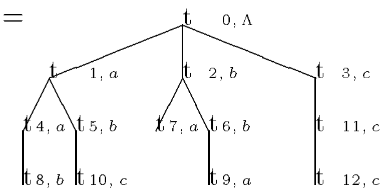
1. T_σ is initialized as a $q+1$-node rooted tree where the root is labeled Λ and numbered 0. The root has q children numbered $1, \ldots, q$. Child i is labeled a_i. The initial dictionary and the empty compression sting *reflect* the null string Λ.

2. Assume that the dictionary and compression string reflect $s_1 \cdots s_i$. Let l be the last defined element in the compression string $\sigma.Z$, and let l' be the last numbered node in T_σ. Let $s_{i+1} \cdots s_j$ be the longest prefix of $s_{i+1} \cdots s_u$ for which there is a node numbered p in T_σ whose string is $s_{i+1}, \ldots, s_j$. Let $\sigma.Z[l+1]$ get value p. If $j \neq u$ then attach a new node numbered $l'+1$ and labeled s_{j+1} as a child of node p in T_σ. The dictionary and compression string now reflect $s_1 \cdots s_j$.

Each element of the compression string represents a *chunk*. The *chunk represented by* $\sigma.Z[i]$ ($\sigma.Z[i]$*'s chunk*) is the string of the node that $\sigma.Z[i]$ points to.

Example: $\Sigma = \{a, b, c\}$, $\quad \sigma = aabbaabbabcccccc$.
$\sigma.Z = 1, 1, 2, 2, 4, 6, 5, 3, 11, 12$.

$T_\sigma =$



Note that the uncompressed string σ has length u while its compressed version $\sigma.Z$ has length n.

Theorem 2.1 (Welch [11]): Given string σ, one can construct T_σ and $\sigma.Z$ in time $O(u)$. Given string $\sigma.Z$, one can construct T_σ and σ in time $O(u)$.

The following theorem and its corollary are crucial to our algorithm. The theorem presents a mapping between the node numbers in the dictionary trie and the chunk locations in the compressed string.

Theorem 2.2 The value assignment to location l in $\sigma.Z$, $1 \le l \le n-1$, causes the definition and creation of node number $l+q$ in the dictionary trie.

Proof: By induction on l. For $l=1$ we have $\sigma.Z[1]=i$, where $s_1=a_i$. The node added has string $s_1 s_2$. This does not appear in the tree and thus causes a new node to be created as a child of node number i. The label of the new node is s_2 and its number is $q+1$. Assume now that node $l+q$ was created when $\sigma.Z[l]$ was assigned its value. Assume also that the trie and compression string reflect $s_1 \cdots s_i$. Let p be the number of the node whose string is $s_{i+1}\cdots s_j$, where $s_{i+1}\cdots s_j$ is the longest prefix of $s_{i+1}\cdots s_n$ that appears as a tree node's string. Therefore there is no dictionary node whose string is $s_{i+1}\cdots s_{j+1}$. Thus a new node labeled s_{j+1} will be created as p's child, and its number is $l+q+1$. ∎

Corollary 2.3 Let $\sigma.Z[i]=l$. The first symbol of $\sigma.Z[i]$'s chunk is the first symbol of $\sigma.Z[l-q]$'s chunk. The last symbol of $\sigma.Z[i]$'s chunk is the first symbol of $\sigma.Z[l-q+1]$'s chunk.

Definition 2.4 Let $P=p_1\cdots p_m$ be a *pattern* string. An *internal substring* $p_i\cdots p_j$ of the pattern is a substring where $i>1$ and $j\le m$. A chunk which is an internal substring is called an *internal chunk*. A chunk is a *prefix chunk* if it ends with a non-empty pattern prefix. The representing prefix of a prefix chunk is the *longest* pattern prefix with which it ends. A chunk is a *suffix chunk* if it begins with a non-empty pattern suffix. Its *representing suffix* is the longest pattern suffix with which it begins.

Lemma 2.5 Let $\sigma.Z$ be a compressed string of length n, P a pattern of length m. There are at most m^2 internal chunks of which at most m chunks are *exact suffixes* of P. There may be as many as n prefix chunks.

Proof: Recall that a chunk is a path from the root to a dictionary trie node. Since there are at most m^2 internal substrings and they must all start at the root, then there may be no more than m^2 internal chunks and of these, at most m are suffixes. Note, however, that all descendents of these m suffixes are what we defined as suffix chunks.

All chunks may be prefix chunks as illustrated by the following example. Let $\sigma = aaa\cdots a$, where a is the first character of the pattern. ∎

3 Basic Algorithm using Explicit Dictionary

As in most pattern matching problems, the algorithm consists of a pattern preprocessing part and a text scanning part (in our case the text is the compressed string $\sigma.Z$). We adopt the notation S_1S_2 to denote the *concatenation* of strings S_1 and S_2. For clarity we will also denote the substring starting at pattern location i and ending at pattern location j as $p_i\cdots p_j$. In reality such substrings are implemented as the pair $\langle i,j\rangle$.

A. Pattern Preprocessing

1. Preprocess the pattern to allow answering the following queries:

   (a) Let S_1 be a pattern prefix, S_2 an internal substring.

   $$Q_1(S_1, S_2) = \text{ the length of the longest pattern prefix that is a suffix of } S_1S_2$$

   (b) Let S_1 be a pattern prefix, S_2 a pattern suffix.

   $$Q_2(S_1, S_2) = \begin{cases} i, & i \text{ is the smallest index of } S_1S_2 \text{ where there is a pattern occurrence;} \\ 0, & \text{there is no pattern occurrence in } S_1S_2. \end{cases}$$

   (c) Let S_1 be an internal substring, $a \in \Sigma$.

   $$Q_3(S_1, a) = \begin{cases} \langle i, j \rangle, & S_1a \text{ is internal substring } p_i \cdots p_j \; j = m \text{ if possible;} \\ \langle 0, 0 \rangle, & S_1a \text{ is not an internal substring.} \end{cases}$$

end Pattern Preprocessing

Time and Space: In Section 4 we will discuss different implementations that allow answering these queries. For the moment let $t_P(m)$ be the pattern preprocessing *time* and $s_P(m)$ be the pattern preprocessing *space*. Let $t_{Q_i}(m)$ be the time to answer query i, respectively.

Text Scanning

The text scanning part has two main components. The dictionary construction and update part, and the pattern search part.

When constucting the dictionary we also introduce and appropriately mark a *prefix flag*, a *suffix flag* and an *internal flag*. In addition, every node stores the first symbol in its string. The prefix flag indicates the length of the representing prefix, the suffix flag indicates the length of the representing suffix and the internal flag indicates the indices of the internal substring that this node represents.

The pattern search part keeps track of the largest pattern prefix that ends with the previous chunk, and then uses the queries to find out if the current chunk extends the prefix or not.

B. Compressed Text Scanning

Initialize: $Prefix \leftarrow \Lambda$.

for $l = 1$ to n do { The sequence below is done for each element of $\sigma.Z$. }

1. *Dictionary Construction and Flagging:*

    (a) Add node number $l + q$ to T_σ. This node is the child of node number $\sigma.Z[l]$. Its label is the first symbol of node $\sigma.Z[l+1]$. (If $\sigma.Z[l+1] = l + q$ then the label is $\sigma.Z[l]$'s first symbol.)

    (b) The first symbol of node number $l + q$ is the first symbol of its parent.

    (c) If $\sigma.Z[l]$ is a suffix node then $l + q$ is a suffix node. Its representing suffix is the same as that of its parent.

    (d) If $\sigma.Z[l]$ is internal node $p_i \cdots p_k$ and the label of $l + q$ is a then mark $l + q$'s internal flag with $Q_3(p_i \cdots p_k, a) = \langle h, j \rangle$. If additionally, node $l + q$ is a suffix node (i.e. $j = m$) then mark its suffix flag with its representing suffix. (This may reset the value marked in (c).)

    (e) If $\sigma.Z[l]$ is a prefix node with representing prefix $p_1 \cdots p_i$ and the label of $l + q$ is a then mark $l + q$'s prefix flag with $Q_1(p_1 \cdots p_i, a)$.

2. *Pattern Search:*

    (a) If $Prefix = \Lambda$ and $\sigma.Z[l]$ is a prefix node with representing prefix $p_1 \cdots p_i$ then $Prefix \leftarrow p_1 \cdots p_i$.

    (b) If $Prefix \neq \Lambda$ and $\sigma.Z[l]$ is a suffix node whose representing suffix is $p_i \cdots p_m$ then if $Q_2(Prefix, p_i \cdots p_m) \neq 0$ we have a pattern appearance.

    (c) If $Prefix \neq \Lambda$ and $\sigma.Z[l]$ is internal node $p_i \cdots p_j$ then $Prefix \leftarrow Q_1(Prefix, p_i \cdots p_j)$, else $\sigma.Z[l]$ is not an internal node and $Prefix \leftarrow$ representing prefix of $\sigma.Z[l]$.

end for loop
end Compressed Text Scanning

Time: $O(n(t_{Q_1}(m) + t_{Q_2}(m) + t_{Q_3}(m)))$.
Space: $O(n)$.

Total Algorithm Time: $O(n(t_{Q_1}(m) + t_{Q_2}(m) + t_{Q_3}(m)) + t_P(m))$.
Total Algorithm Space: $O(n + s_P(m))$.

4 Implementation and Analysis

4.1 $O(n + m^3)$ Time and Space Algorithm

We begin with a simple implementation for queries Q_1, Q_2 and Q_3. This implementation is inefficient but serves as a stepping stone for the better schemes presented in the following sections.

The main idea is to create two $m \times m^2$ tables, N_1 and N_2, where the rows correspond to the m pattern prefixes and the columns correspond to the m^2 internal substrings of the pattern. Denote the j-th pattern substring by $P^{(j)}$.

- $N_1[i,j]$ is the maximum l, $\quad 0 \le l \le i$ for which the length-l suffix of $p_1 \cdots p_i$ concatenated with $P^{(j)}$ is a prefix of the pattern. If there is no suffix of $p_1 \cdots p_i$ that, concatenated with $P^{(j)}$, is a prefix of P, then $l = 0$.
- $N_2[i,j]$ is the length of the maximum suffix of $p_1 \cdots p_i$ that is the prefix of an occurrence of P in $p_1 \cdots p_i P^{(j)}$.

Example:

$P = abcab$, $P^{(j)} = bca$. Then:

- $N_1[4,j] = 1$ because the longest suffix of $abca|bca$ which is a prefix of P has length four and uses only the last letter from $p_1 \cdots p_4$.
- $N_2[4,j] = 4$ because the first occurrence of P in $abca|bca$ uses all four letters of $p_1 \cdots p_4$.

Once tables N_1 and N_2 are constructed, we can answer queries Q_1 and Q_2 in constant time in the following fashion. If $Q_1(p_1 \cdots p_i, P^{(j)}) > |P^{(j)}|$, then $Q_1(p_1 \cdots p_i, P^{(j)}) = N_1[i,j] + |P^{(j)}|$. Otherwise, $(N_1[i,j] = 0)$, check if $P^{(j)}$ is a prefix chunk. As for Q_2, clearly $Q_2[i,j] = i - N_2[i,j] + 1$.

We now show how to construct tables N_1 and N_2 in time and space $O(m^3)$. We will later show a separate method for answering Q_3 queries. We need two data structures as tools for our implementation.

The Knuth-Morris-Pratt (KMP) Automaton: For every prefix $p_1 \cdots p_i$, the KMP automaton provides, in constant time, the border of $p_1 \cdots p_i$, that is, the largest suffix of $p_1 \cdots p_i$ that is also a prefix.

The (Uncompacted) Suffix Trie: A suffix trie ST_S of string S is a trie of all the suffixes of S.

Observations:

1. Every substring of S is a node in ST_S.
2. The leaves that are descendents of node $P^{(j)}$ are exactly the suffixes whose prefix is $P^{(j)}$, i.e. the starting location of these suffixes are exactly the locations in P where $P^{(j)}$ occurs.

Both our tables are similarly constructed. An overview of the table constructing algorithm follows.

Tables N_1 and N_2 Construction Algorithm

1. Construct the KMP Automaton for P.

2. Construct the uncompacted suffix trie for P.

3. Using the suffix trie, fill in all table locations where:

   - $N_1[i,j] = i$ ($i.e.$ where all of $p_1 \cdots p_i P^{(j)}$ is a prefix of P)
   - $N_2[i,j] = i$ ($i.e.$ where the first occurrence of P in $p_1 \cdots p_i P^{(j)}$ begins with p_1).

4. For each table column, fill in the rest of the entries using the KMP automaton.

end Table Construction Algorithm

Implementation and Analysis:

Step 1 is a standard KMP automaton construction that was shown in [8] to be accomplished in time and space $O(m)$. Step 2 can be easily implemented in time and space $O(m^2)$. We need to consider steps 3 and 4.

Step 3: For N_1, by observation 2, in the suffix trie, each leaf descendent from node $P^{(j)}$ corresponds to a suffix of P that begins with $P^{(j)}$. Let there be l such leaves and let the starting locations of these l suffixes be $i_1, \ldots, i_l$. Consider the first suffix with starting location i_1. Clearly, $p_1 \ldots p_{i_1-1} P^{(j)}$ is a prefix of P and $N_1[i_1 - 1, j] = i_1 - 1$. Thus, the locations $i_1 - 1, \ldots, i_l - 1$ are precisely the places where $N_1[i,j] = i$.

For N_2, recall that the compressed text scanning portion of the algorithm only queries Q_2 on a prefix-suffix pair. Thus it is sufficient to construct the N_2 table only for the m substrings of P that are suffixes. If $P^{(j)}$ is a suffix (leaf in the suffix trie) that has starting position i then $N_2[i-1,j] = i-1$. But, in N_2, we have additional nodes where $N_2[i,j] = i$. Denote the length-d prefix of $P^{(j)}$ by $P^{(j|d)}$. Then $N_2[i,j] = i$ at all the nodes where $N_1[i, j|(m-i+1)] = i$ and this latter specification encompasses the first.

It is easy to see that one may construct in time $O(m^2)$ an $m \times m$ table that provides $j|d$, given j and d, in constant time.

Therefore, for every leaf i (suffix $p_i \cdots p_m$), let $P^{(j_1)}, \ldots, P^{(j_r)}$, be the substrings on the path from leaf i to the root. Set all $N_1[i-1, j_l] \leftarrow i-1, \quad l = 1, \ldots, r$. Now for every $[i,j]$ for which $N_1[i, j|(m-i)] = i$, set $N_2[i,j] \leftarrow i$. For each $[i,j]$ where $i + |P^{(j)}| < m$, set $N_2[i,j] \leftarrow 0$.

Time: $O(m^3)$, the time to follow all paths in the suffix trie.

Step 4: This step is identical for tables N_1 and N_2, so we describe it for a "generic" table N. Initially, all table entries where $N[i,j] = i$ are filled. Fix a column j. Repeat the following procedure until there are no more unfilled entries in column j.

Let l be the maximum row for which $N[l,j]$ is unfilled. We want the longest suffix of $p_1 \cdots p_l$ that, concatenated with $P^{(j)}$, has a certain property (is a prefix of P in N_1, and is an occurrence of P, possibly followed by extra symbols, in N_2). In both cases, we need a suffix of $p_1 \cdots p_l$ that is also a prefix of $p_1 \cdots p_l$ and *in addition* has an extra property.

The KMP automaton gives us a list $l_1, \ldots, l_k, \quad l_1 > l_2 > \cdots > l_k = 0$ of prefixes of P that are suffixes of $p_1 \cdots p_l$. (The l_i are the indices of the ends of the prefixes.) We can follow this list until the first l_{i_0}, where $N[l_{i_0}, j]$ is filled. All longer prefixes inherit the same value.

Example:

$p_1 \cdots p_l = aabaabaa$, $P^{(j)} = ab$.

The KMP automaton tells us that prefixes of P that are suffixes of $p_1 \cdots p_l$ end at indices 5, 2 and 1. From step 3, we know that of these, only $l_3 = 1$ has an entry in table N_1 (*e.g.* $N_1[1, j] = 1$). But now we know the entries $N_1[2, j] = N_1[5, j] = N_1[1, j] = 1$.

For all i such that $l_i \geq l_{i_0}$ set

$$N[l_i, j] \leftarrow \begin{cases} N[l_{i_0}, j], & \text{if } l_{i_0} \neq 0; \\ 0, & \text{if } l_{i_0} = 0. \end{cases}$$

Time: Every table entry in the column is filled once. Every entry accesses at most one previously filled entry (the one that stops the KMP path). Therefore the time is $O(m)$ per column, $O(m^3)$ for the entire table.

We still need to implement query $Q_3(P^{(j)}, a)$. $P^{(j)}$ has an outgoing edge labeled a in the suffix trie iff that child is the internal substring desired by the query. Additionally, if that *child* has an outgoing edge labeled with the end-of-string character, then that grandchild is the internal suffix string we desire. The required time and space for query Q_3 is then $O(m^2)$ for preprocessing and constant time per query.

4.2 $O(n + m^2)$ Time and Space Algorithm

The main contributor to the $O(m^3)$ complexity is the size of the N_1 table. Q_2 is a query whose parameters are a prefix and a suffix. Since there are only m prefixes and suffixes, the N_2 table needs only m^2 entries, and thus can be constructed in time $O(m^2)$. This section will show that table N_1 can be reduced to size m^2 by reducing the number of columns to m. We need the following data structure:

The (Compacted) Suffix Tree: The compacted suffix tree of a string is the uncompacted trie where every path of degree-2 nodes (that have one child each) is compacted to a single node.

Observations:

1. The suffix tree of an m-element string is of size $O(m)$.

2. The suffixes of the string are the leaves of the compacted suffix tree.

3. If substring $p_i \cdots p_j$ does not appear explicitly as a suffix tree node then there are two unique nodes b, a corresponding to strings $p_i \cdots p_k, \quad i \leq k < j$ and $p_i \cdots p_l, \quad l > k$ that appear explicitly in the compacted suffix tree with b being the parent of a. Denote $p_i \cdots p_k$ by $B(p_i \cdots p_j)$ and $p_i \cdots p_l$ by $A(p_i \cdots p_j)$. Every pattern location where $p_i \cdots p_j$ appears is *always* followed by $p_{j+1} \cdots p_l$.

Implementation: The compacted suffix tree can be constructed in time $O(m)$ [9, 10]. However, since our tables are of size m^2, we can simply construct the uncompacted suffix trie and then mark every node with more than one child and every leaf as *explicit* nodes. Every unmarked node x points to its first explicit descendent (the node corresponding to $A(x)$). The time and space for this construction is $O(m^2)$.

We are now ready to reduce the size of table N_1. Simply choose only the $O(m)$ columns that represent the explicit nodes. It is clear that the table can be constructed exactly as in the previous section, and therefore that $t_P = s_P = O(m^2)$. The only thing we need to show is that $t_{Q_1} = O(1)$.

The Q_1 queries are used in steps 1.(e) and 2.(c) of the text scanning algorithm. In step 1.(e) the parameters of the query are a prefix and a single alphabet letter. This can be implemented using the KMP automaton construction. The preprocessing time and space is $O(m)$ and the query time is $O(1)$.

In steps 2.(c) we have the assignment $Prefix \leftarrow Q_1(Prefix, p_i \cdots p_j)$. Implement it as follows:

If $N_1(Prefix, A(p_i \cdots p_j)) \neq 0$ then $Prefix \leftarrow N_1(Prefix, A(p_i \cdots p_j)) + j - i + 1$

Else $Prefix \leftarrow$ the longest suffix of $p_i \cdots p_j$ that is a prefix.
{ Recall that the longest suffix that is a prefix is simply the representing prefix of $\sigma.Z[l]$.}

The following theorem proves that even though we are querying the prefix with a different parameter, the result is still correct.

Theorem 4.1 The new value assigned to $Prefix$ after our implementation is exactly $Q_1(Prefix, p_i \cdots p_j)$.

Proof: Let $A(p_i \cdots p_j) = p_i \cdots p_j p_{j+1} \cdots p_{j+k}$. If $Q_1(Prefix, p_i \cdots p_j) \leq j - i + 1$ (the representing prefix of the concatenation is entirely in $p_i \cdots p_j$), then there is no suffix of $Prefix$ that, concatenated with $P_i \cdots p_j$ is a prefix. Afortiori, adding extra elements after $p_i \cdots p_j$ can not help, and $N_1(Prefix, A(p_i \cdots p_j))$ will be 0. Our implementation then correctly chooses the representing prefix of $p_i \cdots p_j$.

If $Q_1(Prefix, p_i \cdots p_j) > j - i + 1$ it means that the largest suffix of $Prefix, p_i \cdots p_j$ that is a prefix extends into the suffix of $Prefix$. By Observation 3, every time $p_i \cdots p_j$ appears in the pattern it is followed by $p_{j+1} \cdots p_{j+k}$. Thus the part of $Q_1(Prefix, p_i \cdots p_j)$ that is a suffix of $Prefix$ is the same for $Q_1(Prefix, p_i \cdots p_j)$ and for $N_1(Prefix, A(p_i \cdots p_j))$. (Note that the next chunk may not start with $p_{i+1} \cdots p_{i+k}$, however, that will be noticed by the algorithm in the next step. The only thing of concern now is to get the correct $Q_1(Prefix, p_i \cdots p_j)$.) ∎

4.3 $O(n \log m + m)$ Time and $O(n+m)$ Space Algorithm

Recall that the KMP automaton provides, for each prefix of a string, a pointer to the longest prefix that is also its suffix (also known as a *border*). The collection of such pointers forms a tree, which we call the *failure tree*. In the failure tree of a string σ, each node represents some prefix of σ, and if $\sigma_1 \dots \sigma_j$ is the parent of $\sigma_1 \dots \sigma_i$, then the edge e between them has label $L(e) = \sigma_{i+1} \dots \sigma_j$. In the failure tree, let $p(v)$ be the parent of v and let $E(v)$ be the edge from $p(v)$ to v.

In [7], Gu, Farach and Beigel (GFB) introduced the *border tree* $T_b^S = (V, E, \mathcal{L})$ as follows:

- V is a subset of the prefixes of S such that $v \in V$ iff either $L(v) = v$ or $L(E(v)) \neq L(E(p(v)))$ in the failure tree of S;
- $E = \{(u,v) | u, v \in V, v \prec u, \neg\exists w \in V, v \prec w \prec u\}$, where $a \prec b$ if a is a suffix of b.
- Edge label $\mathcal{L}(E(u)) = (|u|, |D|, k)$, where D is a non-empty string and n is the maximum integer such that PD^α is a prefix of S for $0 \leq \alpha \leq k$ and such that $P \propto PD \propto \ldots \propto PD^k$. Here, $a \propto b$ if $a \prec b$ and there is no prefix c so that $a \prec c \prec b$. Note that this condition differs from the edge condition in that w was constrained to be in V while c can be any prefix of S.

In [7], it was shown that the border tree of a string can be constructed in linear time and that its depth is logarithmic in the original string. This data structre was in fact defined to answer queries of type Q_1 and Q_2. We can therefore directly apply the GFB solution to get:

Theorem 4.2 ([7]) Using the border tree, we can simultaneously achieve $t_P(m) = O(m)$, $s_P(m) = O(m)$, $t_{Q_1}(m) = O(\log m)$ and $t_{Q_2}(m) = O(\log m)$.

We still need to implement query $Q_3(P^{(j)}, a)$ using $O(m)$ space. This is done similar to the way it is described in section 4.1 but by consulting a compacted suffix tree (instead of the uncompacted suffix trie). For each internal chunk $P^{(j)}$ we save its longest prefix that is a compacted suffix tree node ($B(P^{(j)})$). If $B(P^{(j)}) = P^{(j)}$ then we handle it exactly as in section 4.1. Otherwise, check if a is the expected symbol in the appropriate location of $A(P^{(j)})$.

5 The Tradeoff Idea

5.1 Overview

The algorithm we presented is "almost" optimal in the sense that for small patterns ($m \leq \sqrt{n}$) it runs in linear time and uses linear space. In applications where the data has many inherent redundancies (e.g. FAX data) the compression size is significantly less than the uncompressed data. Our algorithm will serve such applications well. The advantages of this algorithm are not only its asymptotic efficiency but also its simplicity and easy coding. In addition, there are no large constants hidden behind the "big-Oh".

We can not ignore, however, another important applications domain. There exist many systems of small machines with limited memory that need to analyse large data sets. Such systems may not afford $O(n)$ space **in addition to** the compressed file itself. The question we are faced with is: "Are there algorithms that allow pattern matching in the compressed file with $o(n)$ additional space, possibly requiring more time than $O(n)$?" This section presents such a tradeoff.

Our idea is to store only the information that is absolutely necessary for the pattern matching. The information we need is: (i) internal chunks, (ii) suffix chunks, and (iii) prefix chunks. Lemma 2.5 tells us that the number of internal chunks is reasonable ($O(m^2)$). Unfortunately it also tells us that the number of suffix chunks may be prohibitively large ($O(n)$). Our solution to this dilemma is not to store the prefix and suffix chunks, but rather to spend time **computing** whether a chunk is a prefix or suffix chunk.

Notation: Let $\sigma = s_1 \cdots s_n$ be a string. The *reverse of string* σ is the string $\sigma^R = s_n \cdots s_1$.

The outline of the new algorithm is sufficiently similar to Algorithms A and B that we only describe the changes necessary. As in the previous section, we first present the algorithm overview and follow with implementation details and analysis.

A1. Pattern Preprocessing

The only addition to Algorithm A is

Construct a suffix tree ST_{σ^R} of σ^R.

end Pattern Preprocessing

The suffix tree ST_{σ^R} enables finding the representing prefix of a prefix chunk. Note that the leaves of ST_{σ^R} are exactly the reverses of all the prefixes of σ.

B1. Compressed Text Scanning

The overview of the text scanning algorithm is essentially the same as Algorithm B. The main difference is that the dictionary construction and flagging part is entirely discarded. We do, however, add an instruction to process internal chunks.

If $\sigma.Z[l]$ is an internal chunk and $\sigma.Z[l]$ concatenated to the first character of $\sigma.Z[l+1]$ is also an internal chunk, then the node whose number is $l + q$ is also an internal chunk.

end Compressed Text Scanning

We need to show how to implement the following functions without the dictionary trie:

1. Check if a node is an internal chunk.
2. Check if a node is a suffix chunk and find the representing suffix.
3. Check if a node is a prefix chunk and find the representing prefix.

5.2 $O(m^2)$ Extra Space Algorithm

We begin with an implementation that reduces the extra space to $O(m^2)$ but whose time is $O(mn\sqrt{u})$. In the next subsection we improve the time by paying more in space.

Internal Chunks:

We keep a list of the m^2 internal chunks and their T_σ node numbers. This can either be implemented as a balanced search tree or as a perfect hash table. The balanced search tree means $O(\log m)$ lookup time. The hashing scheme is the preferred practical choice and gives constant time per lookup w.h.p.. A dynamic-hashing algorithm with $O(1)$ expected amortized cost per insertion was introduced by [5].

Every element is verified for being an internal chunk. If it is, we need to check if the concatenation of the next chunk's first element still gives us an internal chunk. If the first symbol of the next chunk is known, such a check can be implemented in constant time by consulting the suffix tree. The first element can be found by the following algorithm.

C. Finding First Element of Chunk $\sigma.Z[l]$.

$t \leftarrow l$

while $(\sigma.Z[t] > q)$ do

$\quad t \leftarrow \sigma.Z[t] - q$

end

$First \leftarrow a_{\sigma.Z[t]}$
end Algorithm

Correctness: A recursive application of Corollary 2.3 assures us that the symbol in $First$ is indeed the first symbol of chunk $\sigma.Z[l]$. It is important to note that following the $\sigma.Z$ chunks until the first element is reached really means following the path on the dictionary trie T_σ from a node to the root.

Time: The total time for verifying for every block whether it is an internal block is $O(n \log m)$ (or $O(n)$ if hashing is used). Every block needs to know the first element of the following block. The time for this is the sum of all paths in T_σ. The following lemma shows that this sum is $O(u)$.

Lemma 5.1 The sum of the lengths of all paths in T_σ is $\leq 3u + q$.

Proof: For a node v that appears in $\sigma.Z$, the path from the root to v is the chunk of uncompressed text represented by v. Therefore,

$$\sum (\text{length of path to } v) \leq u$$

where the sum is over all the nodes v in $\sigma.Z$. But T_σ contains other paths as well. Specifically, for every node in $\sigma.Z$, there is potentially one node created in T_σ that is not used in $\sigma.Z$. The sum of path lengths for these nodes is

$$\sum [(\text{length of path to } v) + 1] \leq u + n$$

Finally, there are at most q unused letters in σ and therefore q unused nodes of path length one in T_σ. In summary, the sum of the path lengths is $\leq 2u + n + q \leq 3u + q$.

∎

Suffix Chunks:

Chunks that are *exactly* suffixes are internal chunks. These m special internal chunks are handled as other internal chunks. We need to check for every chunk whether it has a prefix that is a suffix of the pattern. The same algorithm for finding the first element in a chunk can be used to traverse the T_σ path from a node toward the root until an exact suffix is found. The criterion for stopping should be modified to be either reaching an exact suffix (then this is a suffix chunk and the exact suffix is the representing suffix) or reaching a non-suffix symbol (then this is not a suffix chunk).

Time: For similar reasons to the internal chunks the traversals cost $O(u)$. However, for every node in the traversal, we check if this is an exact suffix. This adds a multiplicative $\log m$ factor in the worst case making the time $O(u \log m)$ (but $O(u)$ if hashing scheme is used).

Prefix Chunks:

Assume we are given the elements of a chunk from the last to the first. We can use ST_{σ^R} to find the representing prefix in time $O(\min(m, \text{chunk size}))$. We concentrate, then, on listing the elements of a chunk from back to front.

By corollary 2.3 the last element of $\sigma.Z[l]$ is the first element of $\sigma.Z[l+1]$. We know how to find the first element of a chunk. The problem is that finding the last element of a chunk becomes independent of the chunk size. Conceivably, one may need to traverse very long paths to find the last elements of a short chunk. The following lemma bounds the longest path.

Lemma 5.2 The longest path in $\sigma.Z$ is no longer than $\sqrt{u}$.

Proof: A path of length l represents $\sum_{i=1}^{l} = O(l^2)$ uncompressed text elements. The longest path relative to u will be when the tree is a single path. ∎

Time: If our text is composed predominantly of chunks whose length is greater than m, then we will search for at most m last elements per chunk, and the time for each such element can not exceed $\sqrt{u}$. However, the text may be composed of chunks that are all smaller than m elements, and thus the number of last elements we need to check is no more than u. The time to check each such element is still no greater than $O(\sqrt{u})$. In general, the total time is, then, $O(\min(u^{1.5}, mn\sqrt{u}))$.

Adding up the time for all types of chunks and keeping in mind that $n\sqrt{u} \geq u$ we get:

Total Algorithm Time: $O(mn\sqrt{u} + n\log m + u) = O(mn\sqrt{u})$.

Note that it is easy to convert this algorithm to run with the same time bounds but with only $O(m)$ space. We described the $O(m^2)$ version because it is a special case of our tradeoff and helps clarify the next section.

5.3 $O(\frac{n}{c} + m^2)$ Extra Space Algorithm

The main cost in the time of the $O(m^2)$ space algorithm was accessing elements in the dictionary tree. We will introduce extra pointers in order to shortcut this tree search. The idea is to store data for all the nodes in the ci-th level of T_σ, $i = 1, \ldots, \frac{\text{height}(T_\sigma)}{c}$. The information we keep is:

- node number.
- first element of chunk.
- representing prefix.
- representing suffix.

In addition we keep all information about the internal and exact suffix chunks as described in the previous section. We run the algorithm of section 5.1 with the following implementation:

1. **Internal Chunks:** Check whether the chunk is internal as in section 5.2. Find the first symbol of the next chunk by traversing the path toward the root. However, there is no need to run more than c elements since then we reach a stored node and access that information

in constant time.
Total Time: $O(nc + n\log m)$ ($O(nc)$ using hashing).

2. **Suffix Chunks:** Check whether any of the previous elements in the path toward the root is an exact suffix. This is done as in section 5.2. However, there is no need to check more than at most c nodes, since by then we reach a stored node and access the information in constant time.
Total Time: $O(nc\log m)$ ($O(nc)$ using hashing).

3. **Prefix Chunks:** Assume that $c < m$ and $c < \sqrt{u}$. Within at most c elements on the path toward the root there is a stored node x with representing prefix X. Suppose the substring represented by the path from x to $\sigma.Z[l]$ is internal string $P^{(j)}$. Then $Q_1(X, P^{(j)})$ gives us $\sigma.Z[l]$'s representing prefix in constant time. $P^{(j)}$ has at most c elements, each found in time at most $O(c)$.
Total Time: $O(nc^2)$ for $c < m, \sqrt{u}$. In general the time is $O(n \cdot \min(c, m)\min(c, \sqrt{u}))$.

Total Algorithm Time: The worst case time is $O(nc^2 + n\log m + nc\log m + m^2)$ for $c < m, \sqrt{u}$. The time using hashing is $O(nc^2 + m^2)$.
In general, the worst case time is $O(n \cdot \min(c, m)\min(c, \sqrt{u}) + n\log m + nc\log m + m^2)$ and the time using hashing is $O(n \cdot \min(c, m)\min(c, \sqrt{u}) + m^2)$.

6 Real Life Behavior

We remarked previously that the success of the compression depends on the application domain. The best LZW compression has $n = \sqrt{u}$ (lemma 5.2). Under these circumstances, algorithms A for preprocessing and B for text scanning are probably the best choice.

An interesting domain where LZW compression is used is alphanumeric files of texts and programs. Welch [11] reports studies he has done on the LZW compression of such files. Some of his conclusions:

1. The LZW compression of such files normally yields $n = \frac{u}{2}$.

2. Breaking a large file into smaller ones and compressing separately does not significantly alter the compression size.

The conclusions we can draw from these facts is that the dictionary trie is almost a complete tree. This is due to the fact that for a complete tree the sum of all the paths' lengths ($O(u)$) is $O(n)$. Thus the expected path length is $O(\log n)$. If this is the case then the algorithm we described in section 5.2 will run in expected time $O(n\log^2 n + m^2)$ and use space $O(m^2)$.

In reality, however, one can use heuristics that will further improve the time. The $\log^2 n$ factor comes from the need to find a long prefix. In reality, we will stumble very early and not need to check too many paths per chunk. Furthermore, it will be advisable to first establish the suffix and then check the prefix. Most chunks will disqualify as prefix chunks simply because their length is too short. There will be very few chunks where we will actually look for the prefix. Under the circumstances reported by Welch the time for the compressed matching algorithm described in section 5.2 is likely to be $O(n \log m + m^2)$ and the space $O(m^2)$.

7 Open Problems

In the nascent area of compressed matching it is harder to find a closed problem than an open problem. Start with exact matching in the LZW compression. We presented two implementations, neither of which is optimal. We would like to see a $O(n+m)$ time and space algorithm.

For the model requiring as little extra space as possible, one would like an algorithm with $O(m)$ extra space and better time than $O(mn\sqrt{u})$ that we report. Optimally, we would want the time to be $O(n)$, but anything between the two will be welcome.

In addition to the above two problems, one can go through the compression literature and try to find efficient pattern matching algorithms for the various known compressions. An interesting step in this direction would be the Lempel-Ziv compression.

Finally, for any fixed compression we would like to see more than an algorithm for the simple pattern matching. Important problems to solve are approximate matching, dictionary matching, matching with don't cares, and multidimensional matching. Each of these problems needs efficient parallel algorithms as well.

References

[1] A. Amir and G. Benson. Efficient two dimensional compressed matching. *Proc. of Data Compression Conference, Snow Bird, Utah*, pages 279–288, Mar 1992.

[2] A. Amir and G. Benson. Two-dimensional periodicity and its application. *Proc. of 3rd Symposium on Discrete Algorithms, Orlando, FL*, pages 440–452, Jan 1992.

[3] A. Amir, G. Benson, and M. Farach. Witness-free dueling: The perfect crime! (or how to match in a compressed two-dimensional array). submitted for publication, 1993.

[4] A. Amir, G.M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *Journal of Algorithms*, 13(1):2–32, 1992.

[5] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *Proc. of the 20th Ann. ACM Symp. on Theory of Computing*, pages 524–531, October 1988.

[6] T. Eilam-Tsoreff and U. Vishkin. Matching patterns in a string subject to multilinear trasformations. *Procedings of the International Workshop on Sequences, Combinatorics, Compression, Security and Transmission, Salerno, Italy*, June 1988.

[7] M. Gu, M. Farach, and R. Beigel. An efficient algorithm for dynamic text indexing. Personal communication, submitted for publication.

[8] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

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

[10] P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[11] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17:8–19, 1984.

[12] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, IT-23(3):337–343, 1977.