Pattern Matching in Compressed Raster Images

Renato Pajarola Peter Widmayer

Department of Computer Science
Institute of Theoretical Computer Science
ETH Zurich, Switzerland

Abstract

We study the problem of finding a two-dimensional pattern in a compressed satellite image. We present an algorithm that solves this problem without decompressing the image. The runtime of our algorithm is linear in the sum of the sizes of the compressed image and the pattern, unless the image contains many large parts that match the pattern partially, but not fully — an unlikely situation for satellite images. This study contributes to a better understanding of what operations can be performed directly on compressed raster images.

1 Introduction

The amount of data that is stored in computers has grown substantially over the past few years, and the growth is expected to continue. In particular, satellite images are sent to earth regularly, and they fill computer file archives quickly. This data should therefore be kept in compressed form. With these large amounts of compressed data in computer archives, the need arises to process the data in it's compressed form. Here, processing implies complex operations, such as spatial searches in compressed raster images [15] or pattern matching in compressed files [2, 3, 9]. More details about the role of complex operations in a variety of settings is illustrated in the IEEE Computer special issue on *Finding the Right Image* [10].

For satellite images, we want to be able to quickly match a (small) raster image with a (large) two-dimensional compressed raster image; let us call this the *two-dimensional compressed pattern matching problem.* This problem has been studied for the case in which run-length compression has been applied to the image [2]. In our experience, however, run-length compression is not the method of choice for satellite images; prefix codes such as Huffman's seem to be more efficient in terms of compression ratio and are already used in state of the art compression methods [11, 14].

In this paper, we propose a compressed matching method that is efficient whenever the number of partial matches found in the search process is not very high — a situation that is very likely for satellite images. Our proposal is described in Section 4; it follows the spirit of Bird's [5] proposal for (non-compressed) two-dimensional pattern matching, with an additional twist that aims at the expected case for satellite images. A crucial step in Bird's algorithm is the matching of one-dimensional rows; since there are many in the pattern, this is a multiple one-dimensional pattern matching problem [8]. In the non-compressed case, this problem has been solved by Aho and Corasick [1]. We propose a solution for compressed strings, based on the solution in [1], in Section 3.

In order to arrive at a fair comparison between the compressed and the non-compressed case, we need to apply operations of the same power in both cases. In the latter, a comparison

for equality between two symbols in a given, finite alphabet Σ is counted as one step. To do the same for prefix compressed strings, we compare two bit strings of length $\lceil \log |\Sigma| \rceil$ in one step. This introduces some extra difficulty and complexity into the string matching problem even for a single string. We suggest a solution to this problem in Section 2.

In more detail, we study three problems in the following setting. We are given a finite alphabet Σ, and we count one step for a comparison operation between two symbols from Σ or between two strings of $\lceil \log |\Sigma| \rceil$ bits each. We call $w = \lceil \log |\Sigma| \rceil$ the word size (of a conceptual machine). In their non-compressed form, we have a one-dimensional pattern $P[1..m]$, several patterns $P^i[1..m^i]$ (note that ^{i} is an index, not an exponent), a text $T[1..n]$, a two-dimensional pattern $PP[1..m, 1..m]$, and a two-dimensional image $TT[1..n, 1..n]$. In compressed form, we have $\overline{P}[1..\overline{m}]$, where $\overline{m}$ is the number of w-bit words in $\overline{P}$, and correspondingly $\overline{P^i}[1..\overline{m^i}]$, $\overline{T}[1..\overline{n}]$, $\overline{PP}[1..m, 1..\overline{m}]$ and $\overline{TT}[1..n, 1..\overline{n}]$ (this assumes that a two-dimensional pattern or image is compressed row by row; for details, see Section 4). We study the following three problems:

Compressed String Matching Given the pattern $P[1..m]$ and the compressed text $\overline{T}[1..\overline{n}]$, find all occurrences of P in the uncompressed text T.

Compressed Multiple String Matching Given t patterns $P^1[1..m^1], \ldots, P^t[1..m^t]$, and the compressed text $\overline{T}[1..\overline{n}]$, find all occurrences of any pattern P^i in T.

Compressed Two-Dimensional Pattern Matching Given the two-dimensional pattern $PP[1..m, 1..m]$ and $\overline{TT}[1..n, 1..\overline{n}]$, find all occurrences of PP in TT.

2 Compressed string matching

Here, we describe the problem for one single pattern, and we show a solution which can be extended to multiple strings and is used later in the two-dimensional solution. Clearly, a pattern matching algorithm has to be designed specifically for the particular compression method; the algorithm of [3] shows this for single-pattern matching in Z-compressed files. The pattern matching algorithm that we propose works for popular prefix codes like Huffman's.

2.1 Compression of text and pattern

To find an exact match of a pattern P in its compressed form $\overline{P}$ in the bitstream of a compressed text $\overline{T}$, the search pattern is compressed in the same manner as the text. In the compression method we consider, the compressed form of any character sequence is independent of its position in the text. A compression scheme with this property is called *non-adaptive*. That is, for compressing a pattern $P = T[i..i + l]$, we cannot take into consideration any information on the preceding string $T[0..i - 1]$ or the subsequent string $T[i + l + 1..n]$. Thus, for $m = |P|$, and $P_i = T[i..i + m - 1]$ and $P_j = T[j..j + m - 1]$, where $P_i = P_j$ and $i \neq j$, the compressed bitstreams $\overline{P_i}$ and $\overline{P_j}$ are identical. These restrictions on the compression algorithm imply that e.g. arithmetic coding [6] cannot be used, because the distinction between single symbols in the output gets lost, and the characters $T[0..i - 1]$ influence the coding of the remaining $T[i..n]$. However, common prefix codes such as Rice, Colomb or Huffman [12] can be used.

2.2 Compressed pattern matching

Even for non-adaptive compression methods such as those named above, compressed pattern matching is not trivial. Just finding the bitstream $\overline{P}$ in $\overline{T}$ is not enough, because the symbol

boundaries cannot be identified easily. Figure 1 shows an example with symbol alphabet $\Sigma = \{a, b, c, d, e\}$, mapping $\{a \to 00, b \to 01, c \to 10, d \to 110, e \to 111\}$, text $T = babcada$ and pattern $P = cada$. The first bitwise match of $\overline{P} = 100011000$ from the left within $\overline{T} = 010001100011000$ is not a match with respect to P in T. We can detect this string mismatch, if we can recognize each bit that starts a symbol. We will solve this *symbol boundary problem* by explicitly identifying some of these start bits (by what we call *checkpoints*), thereby breaking the bit sequence into segments within which all start bits can be identified by a scan through this segment only. For more details, see Subsection 2.4.

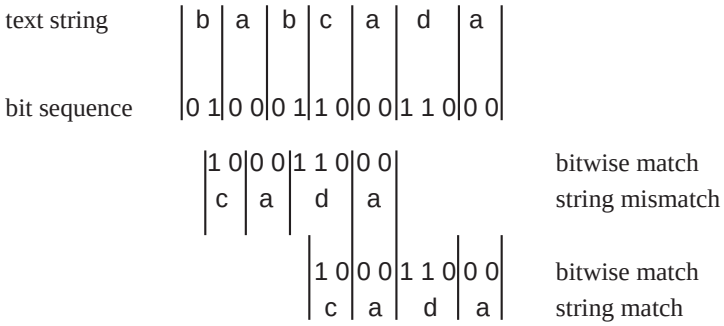


Figure 1: A mismatch and a match

A second problem is that a comparison of the symbols in the compressed alphabet $\overline{\Sigma} = \{0, 1\}$ is less powerful than a comparison of two symbols of Σ. To arrive at a fair computational resource utilization judgement, we want to use the same elementary operations in both cases. To do so, let us represent each symbol in Σ by a fixed length bit string of $w = \lceil \log |\Sigma| \rceil$ bits, and call a bit string of length w a *word*. Now assume that two words can be compared in one step. Due to this aggregation of bits into words, we face another problem: A compressed pattern $\overline{P}$ does not have to start at a word boundary in the compressed text $\overline{T}$; it can start (and end) at any bit position in a word. The next subsection shows a way to handle this *word alignment problem*.

2.3 Word alignment

Figure 2 gives an example for the word alignment problem for 4 bit words with the pattern $P = cadae$ for the compression of Figure 1, that is, for $\overline{P} = 100011000111$.

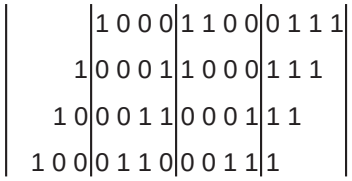


Figure 2: Possible alignments of a pattern

Let $\overline{P_i}$ be the pattern $\overline{P}$ with the i-th symbol of $\overline{P}$ in the first position of a word, $1 \leq i \leq w$, cut to word boundaries. More formally, $\overline{P_i} = \overline{P}[i \,..\, m - ((m - i + 1) \bmod w)]$. For an example, see Figure 3. Let us call $\overline{P_i}$ the *i-th aligned substring* of pattern $\overline{P}$, when the word length w is fixed. An *aligned substring* of $\overline{T}$ is a substring of $\overline{T}$ that starts (and ends) at word boundaries. A match of an aligned substring of $\overline{P}$ with an aligned substring of $\overline{T}$ is called an *aligned match*. We conclude directly:

Lemma 1 *If $\overline{P}$ matches within $\overline{T}$, then at least one of the $\overline{P_i}$, $1 \leq i \leq w$, has an aligned match within $\overline{T}$.*

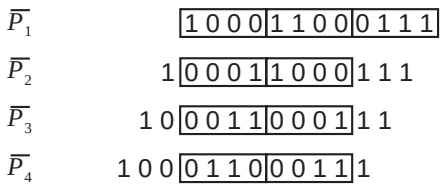
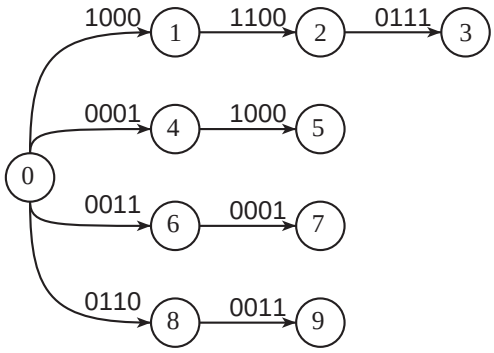


Figure 3: Possible aligned patterns

Of course, an aligned match is only a necessary condition for a bitwise match, because the difference between $\overline{P}$ and $\overline{P_i}$ may or may not match within $\overline{T}$.

Since we limit ourselves to word comparisons, we can find an aligned match by checking $\overline{T}$ for each of the w aligned substrings of $\overline{P}$ — a multiple pattern matching problem that can be solved efficiently with the pattern matching automaton of Aho and Corasick [1], A&C for short (see Figures 4 a), 4 b) and 4 c)). For each aligned match that is found, we perform an additional comparison with each of the two words at the boundary of $\overline{P}$, masked to the corresponding length, using a bit-table (see Figure 4 d)).



a) Goto-graph

state	1	2	3	4	5	6	7	8	9
f(state)	0	0	0	0	1	0	4	0	6

b) Failure-function

state	*output*
3	$\overline{P}_1$
5	$\overline{P}_2$
7	$\overline{P}_3$
9	$\overline{P}_4$

c) Output-function

i	1	2	3	4
	(-,-)	(1,111)	(10,11)	(100,1)

d) Bit-table

Figure 4: functions

2.4 Symbol boundaries

Our solution to the word alignment problem does not imply a solution to the symbol boundary problem. Recall that we need to detect for every bitwise match of $\overline{P}$ in $\overline{T}$, say at positions $\overline{T}[i..i + \overline{m} - 1]$, whether the bit $\overline{T}[i]$ that starts the match in $\overline{T}$ is indeed the first bit of an encoded symbol. If so, we have found a match of P in T; otherwise, we have not. In order to decide for any given bit in $\overline{T}$ whether this bit starts a symbol in T, we could simply

decode $\overline{T}$ from the beginning, using the Huffman tree (or any other prefix coding mechanism). This would, however, require time proportional to the length of the decompressed text T — clearly an undesirable performance. We therefore choose to cut $\overline{T}$ into segments by introducing regularly spaced checkpoints (see Figure 5). Then we decompress the part of $\overline{T}$ that falls in between two adjacent checkpoints only. In more detail, introduce a checkpoint after every d words that code for $\overline{T}$. A checkpoint CP stores the number of pixels coded since the last checkpoint and the offset (in bits) to the beginning of the next pixel in the compressed data stream (see Figure 5). During the scan of the compressed input, the number of pixels coded, and the last checkpoint, can be updated at every checkpoint location. This allows to calculate the current index i and the starting bit of a symbol in $\overline{T}$, beginning at the last checkpoint and processing the bits encountered since then. A reduced version of the decompression method without code generation allows to increment the index at the correct bit-positions, and also to test for starting bits of compressed symbols. For a Huffman (prefix tree) compression method, one simply has to repeatedly follow the path in the tree from the root (start of a symbol) towards a leaf and increment the index i whenever a leaf is encountered.

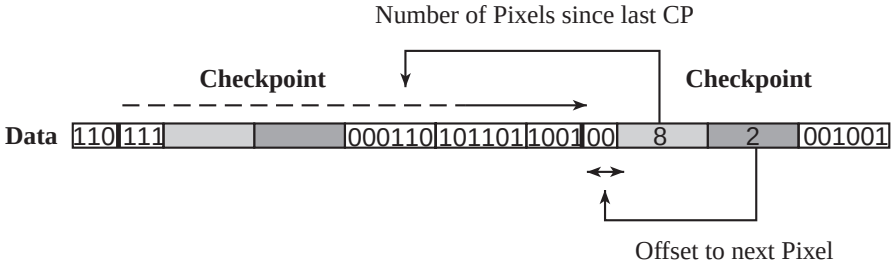


Figure 5: Checkpoints

2.5 Complexity analysis

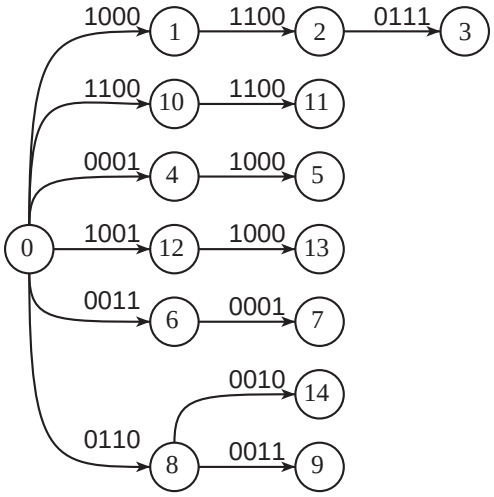
We assume that d, the number of words between any two adjacent checkpoints, and w, the number of bits in a word, are fixed parameters of the implementation.

Theorem 1 *Compressed string matching can be performed in $O(\overline{n}+k)$ time for text searching and $O(m)$ time for pattern preprocessing, where $\overline{n}$ is the number of words in compressed format of text T, m is the length in words of the pattern P, and k is the number of occurrences of any aligned pattern $\overline{P_1}, \ldots, \overline{P_w}$.*

Proof: The pattern P can be compressed in time $O(m)$ into $\overline{P}$. From $\overline{P}$, the aligned patterns $\overline{P_1}, \ldots, \overline{P_w}$ and the bit-table can be produced in time linear in $\overline{m}$. The goto-graph, the failure- and the output-functions can be created in time linear in the sum of the lengths of $\overline{P_1}, \ldots, \overline{P_w}$, which is at most $w\overline{m}$, thus the preprocessing needs $O(\overline{m} + m) = O(m)$.

Finding all matches of one of $\overline{P_1}, \ldots, \overline{P_w}$ in $\overline{T}$ costs no more than $O(\overline{n})$ state transitions in the goto-graph as described in [1]. However, two more checks are necessary for each match of a $\overline{P_i}$ in $\overline{T}$. First, the left- and rightmost bits have to be checked for an exact match of $\overline{P}$ in $\overline{T}$; this can be done with a bit-table similar to Figure 4 d). Second, the start-position of the symbols in $\overline{T}$ has to be checked to match with the beginning of the encountered $\overline{P}$. For this we use the checkpoint method; this needs at most $O(wd)$ steps from the last checkpoint. Therefore, for k occurrences of any aligned pattern $\overline{P_i}$, the work sums up to $O(\overline{n} + k)$. □

Note that in the worst case, k can be close to $w\overline{n}$; in that case, the proposed algorithm is asymptotically slower than the one that decompresses the text and applies the algorithm of



a) Goto-graph

state	1	2	3	4	5	6	7	8	9	10	11	12	13	14
f(state)	0	10	0	0	1	0	4	0	6	0	10	0	1	0

b) Failure-function

state	*output*
3	$\overline{P}^1_1$
5	$\overline{P}^1_2$
7	$\overline{P}^1_3$ $\overline{P}^2_3$
9	$\overline{P}^1_4$
11	$\overline{P}^2_1$
13	$\overline{P}^2_2$
14	$\overline{P}^2_4$

c) Output-function

i \ j	1	2	3	4
1	(-,-)	(1,111)	(10,11)	(100,1)
2	(-,010)	(1,10)	(11,0)	(110,-)

d) Bit-table

Figure 6: functions

Knuth, Morris, and Pratt [13], KMP for short. In many applications, however, we can expect k to be fairly small. In particular, this is the case for compressed satellite image matching, the topic that motivates this study.

3 Multiple compressed string matching

With the goal of applying one-dimensional compressed pattern matching for the two-dimensional case in the spirit of Bird [5], we now turn to the problem of searching for more than one pattern in the compressed text. More precisely, for a given text $\overline{T}$ and t given patterns $P^1, \ldots, P^t$, we want to find all occurrences of any of the P^i in T. This is just an extension to the one-pattern problem. We compress the t different patterns into $\overline{P^1}, \ldots, \overline{P^t}$ and create all possible alignments $\overline{P^1_1}, \ldots, \overline{P^1_w}, \overline{P^2_1}, \ldots, \overline{P^2_w}, \ldots, \overline{P^t_1}, \ldots, \overline{P^t_w}$. For these wt patterns, we create the A&C *goto-graph* as described in [1], and also the corresponding *failure-* and *output-functions*. Using the same symbols and coding as in Section 2.2 with $w = 4$, Figures 6 a), 6 b) and 6 c) show the goto-graph, the failure- and the output-functions for patterns $P^1 =$ *cadae* $\Rightarrow \overline{P^1} = 100011000111$, and $P^2 =$ *dbcac* $\Rightarrow \overline{P^2} = 11001100010$. Therefore, the aligned patterns $\overline{P^1_1} = 1000\ 1100\ 0111$, $\overline{P^1_2} = 0001\ 1000$, $\overline{P^1_3} = 0011\ 0001$, $\overline{P^1_4} = 0110\ 0011$, and $\overline{P^2_1} = 1100\ 1100$, $\overline{P^2_2} = 1001\ 1000$, $\overline{P^2_3} = 0011\ 0001$, $\overline{P^2_4} = 0110\ 0010$ have to be matched.

For every entry in the output-table, which denotes a match of $\overline{P_j^i}$, additional information is stored to check for an exact match of $\overline{P^i}$. A *bit-table* with t rows and w columns can store the left- and rightmost bits for $\overline{P_j^i}$ at position $[i,j]$ $(1 \leq i \leq t \wedge 1 \leq j \leq w)$ as shown in Figure 6 d) for the patterns $\overline{P_1^1}, \ldots, \overline{P_4^2}$ of this section. Each table entry has two fields, containing the left- and rightmost bits which have to be tested. A dash means that nothing (a sequence of length 0) has to be tested on this side.

3.1 Complexity analysis

Theorem 2 *Multiple compressed string matching can be performed in $O(\overline{n}+k)$ time for text searching and $O(m)$ time for pattern preprocessing, where $\overline{n}$ is the number of words of $\overline{T}$, m is the sum of the lengths of the patterns $P^1, \ldots, P^t$, and k is the number of occurrences of any aligned pattern $\overline{P_1^1}, \ldots, \overline{P_w^1}, \overline{P_1^2}, \ldots, \overline{P_w^2}, \ldots, \overline{P_1^t}, \ldots, \overline{P_w^t}$ in $\overline{T}$.*

Proof: Theorem 1 is already based on a modified multiple pattern matching machine. We just need to extend the set of aligned patterns; hence the preprocessing time is linear in the sum of the lengths of the patterns. The text searching step remains the same as in the single compressed string matching algorithm, except that the bit-table is two-dimensional, with one row for each pattern. □

Note that in the worst case, k can be close to $wt\overline{n}$, where t is the number of patterns; again, in the compressed raster image matching case, we expect k to be quite small.

4 Two-dimensional compressed pattern matching

Our two-dimensional compressed pattern matching algorithm aims at raster images. For the algorithmic aspects, a raster image pixel is just a symbol in a (quite large) alphabet. We will argue later why the extra information that our proposal introduces into a compressed raster image does not affect the compression ratio or the runtime of the matching algorithm considerably.

The basic two-dimensional pattern matching algorithm by Bird [5] can be used in conjunction with the proposed modifications of the A&C algorithm of Section 3. The basic idea of Bird is to reduce the two-dimensional problem of searching $PP[1..m, 1..m]$ in $TT[1..n, 1..n]$ to a one-dimensional one. First, (horizontal) matching pattern rows $P^i = PP[i, 1..m]$ of the two-dimensional pattern are searched in the linearized text $TT[1..n^2]$ in the *row-matching* step, and afterwards, the vertical correctness of the matched pattern rows is tested in the *column-matching* step. For this to work, one also needs the column number whenever an exact match of a pattern line occurs.

4.1 Non-compressed pattern matching: A variant of KMP

The work of the column-matching step of Bird's algorithm can be reduced significantly for the average case. In the original algorithm a vector $A[0..n]$ is used, with n the number of columns, to store pattern row id's in such a way that $A[c] = i$ means that the rows $P^1, \ldots, P^{i-1}$ of the pattern PP match the $i-1$ text lines above the current one, with each pattern row ending at column c. This information is enough to use the Knuth, Morris and Pratt algorithm [13], KMP for short, to decide whether a complete pattern really occurred or not, by applying it vertically on every column with the pattern-row id's as the one-dimensional pattern. This is actually done simultaneously on the vector A for all columns of a text line.

Bird ignored the results of the previous row-matching step: The column informations of $A[0..n]$ — which pattern-rows occurred up to the line above the current one — get updated for every column, even if no pattern-row was found. If we store the additional information where the last row-match occurred for column c, we can omit the step of updating $A[c]$ for every column and every row and just update it where a row-match of the pattern really occurred. To do so, let $A[c].pattern$ be the row-id of the pattern which matched ending at column c and $A[c].line$ the text line where this happened; see Algorithm 2 for details. The row-matching step in the two-dimensional situation acts as an *Oracle* for the one-dimensional column-matching step. Algorithm 1 shows this modification of the KMP algorithm, where $last_i$ is equivalent to $A[c].line$, denoting the last occurrence of a symbol of P in T, and j means the same as $A[c].pattern$, the next symbol which has to be matched.

Lemma 2 *Given an* Oracle *that tells at no computational cost which symbol of the pattern P occurs next in the text T and where it occurs, the text searching time complexity of Algorithm 1 is $O(k)$, where k is the number of occurrences of any symbol of the pattern in the text.*

Proof: This property can directly be derived from Algorithm 1, given text $T[1..n]$, pattern $P[1..m]$ and the $next[1..m+1]$ table.

The *Oracle* is denoted as getNextIndexFromOracle$(T, last_i, P)$ below in Algorithm 1, and its computational cost is not counted here. In comparison to the original algorithm in [13], the outer while-loop of Algorithm 1 is performed as many times as getNextIndexFromOracle() returns an index $i \leq n$. For k occurrences of any symbol of pattern P in the text T, getNextIndexFromOracle() yields k different (ascending) indices i and the outer while-loop is performed k times. As in [13], the inner while-loop operation $j := next[j]$ is executed no more often than the outer statement $j := j + 1$. Thus the time complexity of Algorithm 1 is $O(k)$.

The correctness follows from the fact that the *if* $i \neq last_i + 1$ *then* $j := 1$ *endif* catches all situations where in the original algorithm $T[i] \neq P[j], \forall j = 1, \ldots, m$, because j was then set to 0 and incremented afterwards, as it is the case here. □

The preprocessing complexity remains the same as in the original algorithm, $O(m)$, where m is the size of the pattern. In addition to KMP, we make use of an entry $next[m+1]$ to restart the search for the next occurrence of P, the entry is equivalent to the longest suffix of P which is also a prefix of P. We use this KMP-variant in our two-dimensional compressed pattern matching algorithm of Section 4.3 with the goal to reduce the amount of work for comparisons and assignments in the (one-dimensional) column-matching step.

4.2 Compressed pattern matching

The results of Section 2 are used in the compressed row-matching step, where the compressed pattern rows $\overline{P^1}, \ldots, \overline{P^m}$ are used to build the pattern matching machine of Section 3 for multiple compressed patterns. Compression and construction of the goto-, failure- and output-functions are done in a preprocessing step, see Figure 7. After the row-matching step, we have the situation of Figure 8, where the column position information of a pattern-row occurrence is missing, and therefore the vertical alignment of the rows is unknown. The column positions are derived from the checkpoints; they are used to determine the vertical alignment of the pattern rows in the text.

Algorithm 1

$last_i := 0$; *(* index in T of last occurrence of a symbol of P *)*
$j := 1$; *(* next symbol of P to match for in T *)*
$i :=$ getNextIndexFromOracle$(T, last_i, P)$;
while $i \leq n$ **do**
  if $i \neq last_i + 1$ **then**
    $j := 1$
  endif;
  while $j > 0$ **and** $T[i] \neq P[j]$ **do**
    $j := next[j]$ *(* j is decreased in each step *)*
  endwhile;
  if $j = m$ **then**
    reportMatchAt$(i - m)$;
    $j := next[m + 1]$ *(* start for next occurrence of P *)*
  endif;
  $j := j + 1$;
  $last_i := i$;
  $i :=$ getNextIndexFromOracle$(T, last_i, P)$
endwhile;

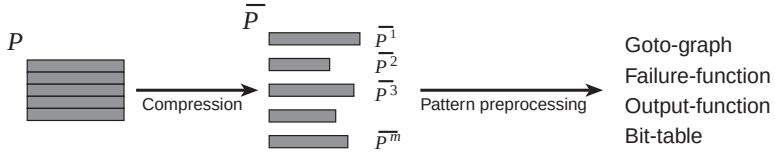


Figure 7: Preprocessing

4.3 Algorithm

Algorithm 2 shows the complete text searching stage of the proposed two-dimensional compressed pattern matching method. The size of the compressed text $\overline{TT}$ is denoted by $\overline{n^2}$, where n is the size of TT in one dimension. The number of rows and the number of columns in the pattern PP is m, and *next* indicates the next-table of the KMP algorithm for the vertical pattern matching of the pattern row id's. The vector a holds the information for the column-matching step. The procedures used in Algorithm 2 are the following:

readNextWord$(\overline{T})$	Returns the next word of the compressed input data stream $\overline{T}$, and updates the current checkpoint *cp* and index *ind* which represents the number of pixels encountered in T up to *cp*.
nextState(x)	Based on the current symbol x, it returns the next state of the goto-graph of the modified pattern matching machine.
checkFirstLastWord(s)	Checks the accordance of the left- and rightmost bits of the pattern row of state s with the text $\overline{T}$. Returns the pattern row id or 0 if the check fails.
calcAlignedColRow(cp, ind)	The alignment of the pattern in $\overline{T}$ is checked and returned in *aligned*, the column and row indices in the original text T are computed and returned in *col* and *row*. These computations

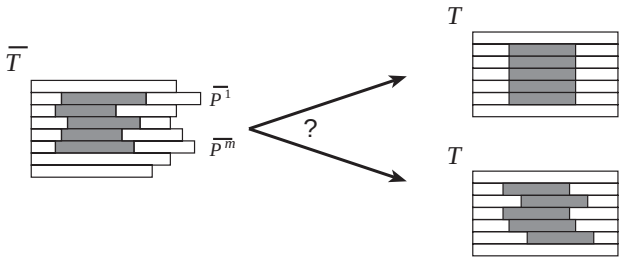


Figure 8: Pattern row alignment

are based on the last checkpoint *cp* and the index *ind* using the simplified decompression algorithm.

reportMatchAt(row, col) outputs an exact match, where the lower right corner of P is at $T[row, col]$.

4.4 Complexity analysis

Algorithm 2 has to preprocess the pattern PP in two ways. First, compress it row by row ($\Rightarrow \overline{P^1}, \ldots, \overline{P^m}$) and generate the functions for the modified A&C compressed pattern matching machine of Section 3. Second, the uncompressed pattern has to be preprocessed the same way as in [5] to get the vertical pattern row id's and the corresponding *next* table of the KMP pattern matching algorithm. Recall that d, the number of words between checkpoints, and w, the number of bits in a word, are fixed parameters of the implementation.

Theorem 3 *Given a two-dimensional compressed text $\overline{TT}$ of size $\overline{n^2}$ and a two-dimensional pattern PP of size m^2, all matches of PP in TT can be found in time $O(\overline{n^2} + k)$, after a preprocessing phase time of $O(m^2)$, and with extra space $O(\overline{m^2} + n)$. Here, k is the number of occurrences of any row P^i of PP in a row of TT.*

Proof: The preprocessing complexity for the row-matching step follows from Theorem 2, as it is exactly multiple compressed pattern matching with the pattern rows $\overline{P^1}, \ldots, \overline{P^m}$ where the sum of the lengths of the compressed pattern rows is $\overline{m^2}$. The preprocessing of the column-matching step remains unchanged w.r.t. [5], thus, with the size m^2 for the pattern PP the total time is in $O(\overline{m^2} + m^2) = O(m^2)$.

The time complexity of Algorithm 2 is mainly determined by the outer for-loop, the readNextWord($\overline{TT}$) and nextState(x) statements, which build together the A&C pattern matching machine described in Section 3. Thus the process of reading the compressed input and determination of the next state is caught in the multiple compressed pattern matching algorithm and therefore follows Theorem 2 with linear time complexity in the text size $\overline{n^2}$. The following if-statements get executed at most as often as a match of an aligned pattern row $\overline{P_j^i}$ ($1 \leq i \leq m \wedge 1 \leq j \leq w$) was found using the multiple pattern matching machine. Therefore, the inner while-loop gets entered at most k times for k occurrences of any aligned pattern row. Together with the previous and following if-statement, the inner while-loop (similar to the inner part of Algorithm 1) performs the KMP variant presented in Section 4.1 and is therefore linear in k, too.

The extra space requirement (apart from storing PP and $\overline{TT}$) of the algorithm is $O(\overline{m^2}+n)$ because the vector a has to hold the pattern id and line information for all n columns of the

Algorithm 2

```
for i = 1 to n^2 do
    x := readNextWord(T);          (* updates also cp and ind *)
    s := nextState(x);          (* A&C pattern matching machine *)
    if s = "terminal state" then
        p := checkFirstLastWord(s);
        if p > 0 then (* a match of a P_i was found in T *)
            (aligned, col, row) := calcAlignedColRow(cp, ind);
            (* continue if symbol start coincides within T *)
            (* and if pattern is not wrapped between text rows *)
            if aligned and col ≥ m then
                l := a[col].line;
                if l ≠ row − 1 then
                    a[col].pattern = 1
                endif;
                j := a[col].pattern;
                while j > 0 and p ≠ P[j] do
                    j := next[j]
                endwhile;
                if j = m then
                    reportMatchAt(row, col);
                    j := next[m + 1]          (* reset for next search *)
                endif;
                a[col].pattern := j + 1;
                a[col].line := row
            endif
        endif
    endif
endfor;
```

text TT and the row-matching step needs $O(\overline{m^2})$ space for the goto-, failure- and output-functions. □

The proposed algorithm works only if the compressed text $\overline{TT}$ contains checkpoints. Too many checkpoints affect the compression ratio, too few affect the runtime of the algorithm. Let us now show that for practical situations, there is enough room for good choices of the number of checkpoints. The time to calculate the alignment value *aligned*, and the row and column indices *row* and *col* just depends on the parameter d and is independent of the input text or pattern. The additional space requirement for these checkpoints is quite low. For an image with a resolution of r bits, where r is typically 8, 16, or 24, alphabet size $|\Sigma| = 2^r$, and with d words in between any two checkpoints, the number of bits used to describe a single checkpoint is $\log wd + \log|\Sigma|$, because no more than wd pixels can be coded since the last checkpoint, and the maximum offset is equal to the longest code length of any symbol, which is $|\Sigma|$ for the prefix codes regarded here. Therefore, for an 8 bit greyscale image with $d = 32$ and $w = 8$, the additional space for one checkpoint is $\log(8 \cdot 32) + \log 256 = 16$ bits, which amounts to 6.25% of the compressed file size (32 data bytes followed by 2 checkpoint bytes). If we raise the number of bytes between checkpoints to $d \geq 512$, the size of a checkpoint description increases to 24 bits; however, the extra information will only enlarge the file size by no more than 0.6% (512 data bytes followed by 3 checkpoint bytes).

5 Conclusion

We have shown how to find all matches of a pattern in a compressed satellite image in linear time, except for the case in which parts of the pattern match, but the whole does not. Even though this case is a very unlikely one in satellite imagery, the problem of finding an algorithm with a linear performance even in the worst case remains open. However, average runtime could be improved further by replacing the multiple string matching machine from Aho-Corasick with the Commentz-Walter [7] method. Also using [4] instead of Bird's 2D algorithm is possible, although this would increase the row and column calculations using checkpoints for setting up in the compressed image for the *checkmatch* procedure described in [4].

In the course of this algorithm, we have studied the details of compressed string matching. In order to operate on the machine word level (and not the bit level), we introduced extra data — the checkpoints — into the compressed text. Even though this extra data does not affect performance a lot, it is an interesting question whether matching in a purely prefix compressed text can also be achieved in time linear in the size of the compressed text.

Note that, even though in the description in this paper, the two-dimensional image and the two-dimensional pattern are both squares, this is not an essential restriction; the algorithm operates just as efficiently for any other rectangular pattern and image.

Acknowledgement

We would like to thank the unknown referee for providing reference [4].

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[2] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In J. A. Storer and J. H. Reif, editors, *Proc. Data Compression Conference*, pages 279–288. IEEE, 1992.

[3] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. In *Proc. of the 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 705–714. ACM, 1994.

[4] R. Baeza-Yates and M. Régnier. Fast two-dimensional pattern matching. *Information Processing Letters*, 1(45):51–57, January 1993.

[5] R. S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, October 1977.

[6] J. G. Cleary, R. M. Neal, and I. H. Witten. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[7] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. of the 6th International Colloquium on Automata, Languages and Programming ICALP*, volume 71 of *Lecture Notes in Computer Science*, pages 118–132, Berlin, 1979. Springer-Verlag.

[8] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, New York, Oxford, 1994.

[9] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. In *Proc. Symposium on Theory of Computing*, pages 703–712, 1995.

[10] V. N. Gudivada and V. V. Raghavan. Content-based image retrieval systems. *IEEE Computer*, 28(9):18–22, September 1995.

[11] P. G. Howard and J. S. Vitter. Fast and efficient lossless image compression. In J. A. Storer and J. H. Reif, editors, *Proc. Data Compression Conference*, pages 351–360. IEEE, 1993.

[12] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proc. Inst. Electr. Radio Eng.*, pages 1098–1101, 1952.

[13] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, June 1977.

[14] G. Langdon, A. Gulati, and E. Seiler. On the JPEG model for lossless image compression. In J. A. Storer and J. H. Reif, editors, *Proc. Data Compression Conference*, pages 172–180. IEEE, 1992.

[15] R. Pajarola and P. Widmayer. Spatial queries on compressed raster images: How to get the best of both worlds. Technical Report 240, Dept. of Computer Science, ETH Zürich, 1995. ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/2xx/240.ps.