An Efficient Pattern Matching Algorithm

Azzam Sleit (previously, Azzam Ibrahim), Wesam AlMobaideen, Aladdin H. Baarah and Adel H. Abusitta
Department of Computer Science, King Abdullah II School for Information Technology College,
University of Jordan, Amman, Jordan

Abstract: In this study, we present an efficient algorithm for pattern matching based on the combination of
hashing and search trees. The proposed solution is classified as an offline algorithm. Although, this study
demonstrates the merits of the technique for text matching, it can be utilized for various forms of digital data
including images, audio and video. The performance superiority of the proposed solution is validated
analytically and experimentally.

Key words: Pattern, matching, hashing, text matching, preprocessing

INTRODUCTION

Pattern matching is a basic problem in computer
science which occurs naturally as part of data processing,
information retrieval and speech recognition. String
(or text) matching is a special case of pattern matching,
where the pattern is described by a finite sequence of
symbols (or alphabet) Σ. It consists of finding one or all
the occurrences of a pattern P of length m in a pattern
database T consisting on n patterns, where m and n > 0.
Both P and T are built over the same alphabet Σ.

Numerous solutions to the pattern matching problem
have been proposed (Aho, 1990). Pattern matching
algorithms are classified into online and offline solutions.
Online solutions are dynamic and do not require a priori
knowledge of the patterns database T. Preprocessing may
be performed on P. In general, an online algorithm
consists of two phases: The preprocessing phase of P
and the search phase of P in T. During the preprocessing
phase, a data structure X is constructed which is
usually proportional to the length of the pattern and
details vary for different algorithms. The search phase
uses the data structure X and tries to quickly determine
if the pattern occurs in the text. This phase is typically
based on four different approaches including classical,
suffix automata, bit-parallelism and hashing. However,
online solutions are based on preprocessing activities
performed on the patterns database T in preparation
for the matching process. This study proposes a hash-
table based solution for the pattern matching problem.

Classical text matching algorithms are based on
character comparisons. The Brute-Force algorithm
(Cormen et al., 2003) (in short, BF algorithm) performs
character comparisons between a character in the text P
and each character in the pattern database from left to
right. In any case, after a mismatch or a complete match
of the entire pattern it shifts exactly one position to the right.
It requires no preprocessing phase and no extra space.
The BF algorithm has O (mn) worse-case time complexity.
The average number of character comparisons is
n(1+1/|Σ|). The Knuth-Morris-Pratt algorithm (in short,
KMP) (Knuth et al., 1977), which was the first linear time
string matching algorithm discovered, performs character
comparisons from left to right. In case of mismatch, it uses
the knowledge of the previous characters in order to
compute the next position of the pattern to use. Boyer-
Moore algorithm (also recognized as BM) (Boyer and
Moore, 1977) is known to be very fast in practice. It
performs character comparisons between a character in
the text and a character in the pattern database from right
to left. After a mismatch or a complete match of the entire
pattern, it uses two shift heuristics to shift the pattern to
the right. Finally, the expected performance of the BM
algorithm is sub linear requiring about n/m character
comparisons on average. The Boyer-Moore-Horspool
(BMH) algorithm does not use the match heuristic
(Horspool, 1980). In case of mismatch or match of the
pattern, the length of the shift is maximized by using
only the occurrence heuristic for the text character
corresponding to the rightmost pattern character (and
not for the text character where the mismatch occurred).
The Quick Search (QS) algorithm performs character
comparisons from left to right from the leftmost pattern
character and in case of mismatch it computes the shift
with the occurrence heuristic for the first text character
after the last pattern character by the time of mismatch
(Sunday, 1990). The preprocessing and searching time of

Corresponding Author: Azzam Sleit (previously, Azzam Ibrahim), Department of Computer Science,
King Abdullah II School for Information Technology College, University of Jordan, Amman, Jordan
2691
Table 1: Time and space requirements for various matching algorithms

<table>
<thead>
<tr>
<th>Category</th>
<th>Algorithm</th>
<th>Preprocessing phase (Time requirements)</th>
<th>Searching phase (Time requirements)</th>
<th>Space requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical approach</td>
<td>BF</td>
<td>—</td>
<td>O (m)</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>KMP</td>
<td>O (m)</td>
<td>O (m)</td>
<td>O (m)</td>
</tr>
<tr>
<td></td>
<td>BM</td>
<td>O (m +</td>
<td>E</td>
<td>)</td>
</tr>
<tr>
<td></td>
<td>BMH</td>
<td>O (m +</td>
<td>E</td>
<td>)</td>
</tr>
<tr>
<td></td>
<td>QS</td>
<td>O (m +</td>
<td>E</td>
<td>)</td>
</tr>
<tr>
<td></td>
<td>BMS</td>
<td>O (m +</td>
<td>E</td>
<td>)</td>
</tr>
<tr>
<td></td>
<td>THM</td>
<td>O (m +</td>
<td>E</td>
<td>)</td>
</tr>
<tr>
<td>Suffix automata</td>
<td>RF</td>
<td>O (m)</td>
<td>O (m)</td>
<td>O (m)</td>
</tr>
<tr>
<td>Bit parallelism</td>
<td>SO</td>
<td>O (m +</td>
<td>E</td>
<td>)</td>
</tr>
<tr>
<td></td>
<td>BNDM</td>
<td>O (m +</td>
<td>E</td>
<td>)</td>
</tr>
<tr>
<td>Hashing approach</td>
<td>KR</td>
<td>O (m)</td>
<td>O (m)</td>
<td>Constant</td>
</tr>
</tbody>
</table>

The QS algorithm is same as the BMH algorithm. The Boyer-Moore-Smith (in short, BMS) algorithm, noticed that computing the shift with the text character just next the rightmost text character gives sometimes shorter shift than using the rightmost text character (Smith, 1991). The Turbo-BM (in short, TBM) (Crochemore et al., 1994) introduced a variation for the BM algorithm. It consists of remembering substring of the text that matched a suffix of the pattern during the last character comparisons (and only if a good suffix shift has been performed).

The second category is based on the suffix automata approach which uses the suffix automaton data structure that recognizes all the suffixes of the pattern. The Reverse Factor (in short, RF) algorithm performs the characters of the text from right to left using the smallest suffix automaton of the reverse pattern (Leecroq, 1992).

Bit parallelism is the third category which uses the intrinsic parallelism of the bit manipulations inside computer words to perform many operations in parallel (whose number of bits in the computer word we denote w). This technique has become a general way to simulate simple Nondeterministic Finite Automata (NFA) instead of converting them to deterministic. The basic idea of the first Shift-Or (in short, SO) algorithm (Baeza-Yates and Gonnet, 1992), is to represent the state of the search as a number and each search step costs a small number of arithmetic and logical operations, provided that the numbers are large enough to represent all possible states of the search. Another algorithm of the bit parallelism activity is called Backward Nondeterministic Matching (BNDM) (Navarro and Raffinot, 1998). This algorithm uses a nondeterministic suffix automaton that is simulated using bit parallelism.

The fourth category of pattern matching is based on hashing. The Karp–Rabin (in short, KR) algorithm is based on hashing. Hashing provides a simple method to avoid a quadratic number of character comparisons in most practical situations (Cormen et al., 2003). The main idea of the KR algorithm is to compute the signature or hashing function of each possible m-character substring in the text and check if it is equal to the signature function of the pattern. Table 1 summarizes the algorithmic run-time requirements of the previous algorithms taking into account preprocessing phase, search phase and space.

**PATTERN MATCHING TECHNIQUE**

Here it is introduced a two-phase hash-table based pattern matching technique. In the first phase, an index structure is created for the database to be used during the pattern matching phase (i.e., second phase). Although, the proposed technique is suitable for the general pattern matching problem, this study will investigate its merits particularly with respect to text matching.

**The index structure:** Our index structure is a two-dimensional hash table H with dimension |Σ| X m, where Σ is the set of alphabet constituting the patterns database and m is the maximum number of alphabet symbols that a pattern can have in database. A pattern P is expected to be found in H(I, j) if and only if |P| = j and I is the outcome of function F when applied on the first symbol of pattern P. For the purpose of indexing a database of text strings, the function F can be defined as the corresponding ASCII code of the first character of the pattern. For every H(I, j), there will be several database patterns (strings) which will be organized as a binary search tree. In other words, all patterns (strings) starting with the same alphabet symbol (character) will be mapped to the same cell in the two-dimensional hash table. Moreover, those strings mapped into the same cell H(I, j) in H will be organized into the same binary search tree based on the following key calculation rule:

\[
\text{Key}(P) = 1 \cdot \text{ASCII}(p_1) + 2 \cdot \text{ASCII}(p_2) + 3 \cdot \text{ASCII}(p_3) + \ldots + j \cdot \text{ASCII}(p_j)
\]  

where, P is the pattern \(<p_1, p_2, p_3, \ldots, p_j>\).

For a database of N English strings, the following issues must be taken into account during the preprocessing phase:

- For every string in the database, three things must be calculated:
• The ASCII code corresponding to the first character of the string (i.e., row I of the hash table).
• The length of the string (i.e., column j of the hash table).
• The corresponding key of the string based on formula (1).
• Create a one balanced binary tree for each hash table cell H(I, j) based on the keys calculated in 1(a) for the strings mapped into H(I, j).

Example: Consider the following database consisting of twelve patterns.

the Rabin Karp algorithm seeks to speed up searching in the text.

Table 2 presents the outcome of step 1 of the preprocessing activities required for building the index. The table summaries information, which will be inserted into the index structure. Figure 1 displays the complete index including the hash table and the binary trees corresponding to the database.

Pattern matching algorithm: In order to search for a pattern \( P = \langle p_1, p_2, p_3, \ldots, p_l \rangle \) in the database using the index, the following must be calculated:

• The ASCII code of the first character in \( P \); namely: \( p_1 \), denoted by \( L \).
• The length of the \( P \), denoted by \( m \).
• The sum of the ASCII code of the characters that form the pattern using formula (1), denoted \( \text{sumASCII} \).

The search process for the pattern \( P \) featured by \( m \), \( L \) and \( \text{sumASCII} \) is as follows:

Table 2: Preprocessing activity for building the index on the example database

<table>
<thead>
<tr>
<th>Word</th>
<th>ASCII code of first character (L)</th>
<th>Position in the text (index)</th>
<th>Length (m)</th>
<th>sumASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>116</td>
<td>1</td>
<td>3</td>
<td>321</td>
</tr>
<tr>
<td>Rabin</td>
<td>82</td>
<td>5</td>
<td>5</td>
<td>495</td>
</tr>
<tr>
<td>Karp</td>
<td>75</td>
<td>11</td>
<td>4</td>
<td>398</td>
</tr>
<tr>
<td>algorithm</td>
<td>97</td>
<td>16</td>
<td>9</td>
<td>867</td>
</tr>
<tr>
<td>seeks</td>
<td>115</td>
<td>26</td>
<td>5</td>
<td>589</td>
</tr>
<tr>
<td>to</td>
<td>116</td>
<td>32</td>
<td>2</td>
<td>227</td>
</tr>
<tr>
<td>speed</td>
<td>115</td>
<td>35</td>
<td>5</td>
<td>560</td>
</tr>
<tr>
<td>up</td>
<td>117</td>
<td>41</td>
<td>2</td>
<td>229</td>
</tr>
<tr>
<td>searching</td>
<td>115</td>
<td>44</td>
<td>9</td>
<td>948</td>
</tr>
<tr>
<td>in</td>
<td>105</td>
<td>54</td>
<td>2</td>
<td>215</td>
</tr>
<tr>
<td>the</td>
<td>116</td>
<td>57</td>
<td>3</td>
<td>321</td>
</tr>
<tr>
<td>text</td>
<td>116</td>
<td>61</td>
<td>4</td>
<td>453</td>
</tr>
</tbody>
</table>

Fig. 1: Index structure for example database

• Access \( H(L, m) \) to get the pointer to the binary search tree which may contain \( P \).
• Use \( \text{sumASCII} \) as the search key to search the binary search tree. If the value of \( \text{sumASCII} \) is found, the corresponding pointer is utilized to locate \( P \) in the database.
• Additional occurrences of \( P \) may be retrieved by following the left child pointer of the node until a node with a different search key has been found.

ANALYSIS

Now it is analyzed the performance of the proposed algorithm worse and average case behavior. It is obvious that \( H \) has \( |\Sigma| \) rows and \( m \) columns, where \( |\Sigma| \) is the size of the alphabet and \( m \) is the maximum number of symbols that a pattern can have. Accordingly, there are \( |\Sigma|^m \) cells in the hash table.

Average case behavior of search: For a database of \( N \) patterns, assume that the patterns are equally distributed onto the hash table cells. It is expected that our search algorithm will have its typical behavior.

Lemma: The average run-time of the search algorithm is \( O(|\log N|/(|\Sigma|^m)) \).

Proof: The proof stems from the fact that for a database of \( N \) patterns, it is expected that \( N/(|\Sigma|^m) \) patterns are mapped
into each cell of \( H \). Since the preprocessing phase has complete a priori knowledge of all patterns and their mappings, balanced binary search trees will be created with logarithmic height. Consequently, the average run-time of the search algorithm is \( O(\log N/(\Sigma^* m)) \).

**Worst case behavior of search:** It is not expected that all \( N \) patterns of the database are mapped into one cell of the hash table. The search algorithm will have its worst performance.

**Lemma:** The search algorithm does not behave any worse than \( O(\log N) \).

**Proof:** Assuming that all \( N \) patterns of the database are mapped into one cell, the cell will point to a large binary tree with height \( \log N \). Consequently, the search algorithm does not behave any worse than the time complexity \( O(\log N) \).

**Space requirements:** It is obvious that most of the required space for the proposed techniques is attributed to the binary search trees.

**Lemma:** The space requirements of the proposed technique is \( O(N+|\Sigma|^* m) \).

**Proof:** The index structure consists of a hash table and binary trees. The hash table requires \( d^* (|\Sigma|^* m) \) bytes, where \( d \) is the number of bytes needed for each cell. Concerning the binary trees, each pattern in the database will be stored in a node of a binary search tree. Consequently, \( c^* N \) bytes are required for the binary search trees, where \( c \) is the number of bytes required to store the contents of a node in the tree. Therefore, the index requires \( (d^* (|\Sigma|^* m) + c^* N) \) bytes or simply, \( O(N + |\Sigma|^* m) \).

The suffix array is an offline mechanism which can be obtained by collecting the leaves of the suffix tree in left-to-right order (assuming that the children of the suffix tree nodes are lexicographically ordered left-to-right by the edge labels). However, it is much more practical to build them directly. In principle, any comparison-based sorting algorithm can be used, as it is a matter of sorting the \( n \) suffixes of the text, but this could be costly especially if there are long repeated substrings within the text. There are several more sophisticated algorithms, from the original \( O(	ext{log } n) \) time (Manber and Myers, 1993) to the latest \( O(n) \) time algorithms (Kim et al., 2005). In practice, the best current algorithms are not linear-time ones (Manzini and Ferragina, 2004). Gonnont et al. (1992) demonstrated that suffix arrays are more powerful than offline indexing based on inverted files.

From the previous discussions, it is clear that the proposed solution is more superior to all the algorithms presented in Table 1 in addition to the offline suffix array in terms of the speed of pattern matching. However, the space required for the hash table and binary search trees tends to be higher than space requirements of the other algorithms. We believe that our proposed technique is well suited for large databases of patterns. For such environments, it is quite natural to define additional data structures such as indexes for the benefit of better search response time. Furthermore, binary search trees can be replaced by secondary-storage index structures such as B*-Trees.

**RESULTS AND DISCUSSION**

In order to experimentally assess the performance of our pattern matching algorithm relative to other algorithms, we selected two English dictionaries; namely: Mawrid and Wafi to represent two independent databases. Mawrid and Wafi dictionaries are of size 1, 012, 015 and 2, 325, 663 characters long, respectively. The speed of pattern matching (in terms of number of comparisons) was compared to those of Boyer-Moore (BM), Quick Search (QS), Reverse Colussi (RC) and Apostolico-Giancarlo (AG) algorithms. The BM algorithm is known to be very fast in applications while the QS algorithm is fast in practice for short patterns and long alphabets. Both of the RC and AG algorithms are variations of the BM algorithm.

Present experiment considered pattern lengths ranging from 2 to 15. For every pattern length in the specified range, we randomly picked 1000 patterns which actually exist in the databases and ran our implementation for all five algorithms. For every algorithm and per pattern length in the range, the average number of comparisons was accumulated. The average numbers of comparisons for each pattern length is shown in Fig. 2 and 3. The performances of all five algorithms (in terms of number of comparisons) for the selected pattern lengths are displayed in Fig. 2 (Mawrid database) and 3 (Wafi database). The proposed solution in this study outperforms the other four algorithms especially for the larger database (i.e., Wafi). This is also true when the number of patterns of a specific length is large. This happens for pattern length ranging between 5 and 15 since English words of lengths between 5 and 15 are very frequent.
CONCLUSION
This research presents a new solution for pattern matching using an auxiliary index data structure. The runtime performance of the proposed pattern Matching algorithm is logarithmic which is far better than the exiting online and offline algorithms which tend to have linear time complexities at best. The study presented a mathematical analysis for the average and worse case behavior of the proposed solution. Moreover, four algorithms were experimentally compared to the proposed algorithm in terms of the average number of comparisons per pattern length. The experimental results demonstrate superiority of the algorithm for large databases with high frequency pattern lengths.

REFERENCES