American Journal of Applied Sciences 6 (1): 78-88, 2008
ISSN 1546-9239
© 2008 Science Publications

Efficient Processing for Binary Submatrix Matching

Azzam Sleit, Wesam AlMobaideen, Mohammad Qatawneh, Heba Saadeh
Department of Computer Science, King Abdulla II School for Information Technology
University of Jordan, Amman, Jordan

Abstract: The heavy demand for large volumes of digital data has increased the interest in matrix-like representation. Matrices are well organized data structures which are suitable to store uniform data in order to simplify data access and manipulation. For several applications, the need is critical to efficiently search for a specific pattern in matrix structures. A pattern can be represented as an n-dimensional matrix which can be searched for within other larger n-dimensional matrices. This query will be referred to as matrix submatching. In this paper, we present and compare two algorithms for binary matrix submatching on the basis of time requirement. The first algorithm is a naive brute force approach with $O(n^2m^2)$ time requirement. The second approach is based on chain code transformation which reduces the sizes of matrices resulting in less time requirement.

Key words: Matrix submatching, brute-force search, submatrix matching, chain code, transformation

INTRODUCTION

The importance of matrices comes from their wide range of applications in various areas such as image processing, geographic information systems, speech recognition, document classification, and bioengineering[1,6,12]. Operations on matrices are at the heart of scientific computing. Efficient algorithms for working with matrices are therefore of considerable practical interest. Matrix operations such as multiplication received much research attention[2,3,5]. In 1992, Shen and Hu studied a new kind of relationship between matrices, namely, approximate submatrix matching (ASM). Given two n x m matrices A and B, find a k×l submatrix in A and another k×1 submatrix in B such that their difference is minimized under a certain measure function. They discussed the ASM problem under two typical measure functions, namely, convolution and Euclidean distance[10]. In 2006, Koyuterk and Grama built a software system, called PROXIMUS, for error-bounded approximation of high-dimensional binary attributed datasets based on nonorthogonal decomposition of binary matrices. This tool can be used for analyzing data arising in a variety of domains ranging from commercial to scientific applications. Using a combination of innovative algorithms, novel data structures, and efficient implementation, PROXIMUS demonstrated rather good accuracy, performance, and scalability to large datasets. The technique was experimented on diverse applications in association with rule mining and DNA microarray analysis[8].

The matrix containment or submatching problem received almost no attention in the literature. We believe that the matrix submatching problem is quite important and deserves attention from researchers due to the vast applications that may require such functionality. This article chooses to focus on defining and solving the exact binary submatching problem and will certainly pave the way for future research activities leading to non-exact general matrix submatching. The following definition formally presents the MSM function which accepts two matrices A and B and returns a set of (i, j) locations in matrix A where matrix B completely appears in A starting at raw i and column j of matrix A. Matrix B may appear zero or more times in A.

Definition: Given two matrices A: nXn and B: mXm, such that $m \leq n$, MSM (A, B) is the set of all occurrences of B in A. Formally, for $1 \leq i \leq n$ and $1 \leq j \leq m$,

MSM(A, B)= {A(i, j): A(i, j) = B(1, 1), A(i, j+1) = B(1, 2), …, A (i, j+m-1) = B (1, m), and
A (i+1, j) = B (2, 1), A (i+1, j+1) = B (2, 2), …, A (i+1, j+m-1) = B (2, m), and
A (i+2, j) = B (3, 1), A (i+2, j+1) = B (3, 2), …, A (i+2, j+m-1) = B (3, m), and
… A (i+m-1, j) = B (m, 1), A (i+m-1, j+1) = B (m, 2), …, A (i+m-1, j+m-1) = B(m, m)}

Corresponding Author: Azzam Sleit, Department of Computer Science, King Abdulla II School for Information Technology, University of Jordan, Amman, Jordan

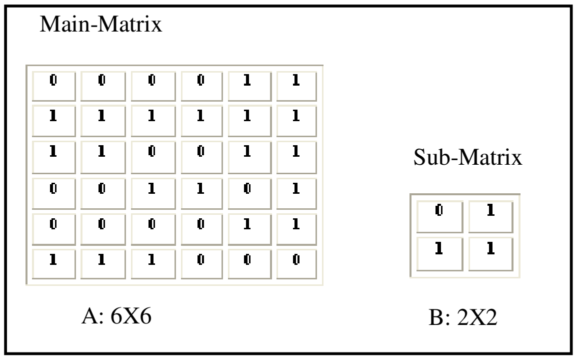
BRUTE-FORCE METHOD

The conventional algorithmic solution for the search problem is to sequentially search for a particular pattern until the pattern has either been found or the search space exhausted without any match. This approach is typically referred to as brute-force search or exhaustive search[2,4,9]. Brute-force search is simple to implement, and will always find a solution if it exists. Brute-force search has the advantage that it requires no imagination or cleverness. Fig. 1 describes a brute-force algorithm for the matrix submatching problem. The algorithm expects two matrices A:nXn and B:mXm where $m \leq n$ as input, while A is the main matrix, B is the submatrix. The algorithm goes through the first n-m+1 rows of the main matrix and for each row it scans the first n-m+1 columns in order to find the upper left corners of potential matches. For each element of the $(n-m+1)^2$ elements in the main matrix, the algorithm performs at least one comparison and at most m^2 comparisons with the elements of the submatrix. It is obvious that the Brute-force algorithm requires at least $(n-m+1)^2$ (i.e. $\Omega(n^2)$) and at most $m^2(n-m+1)^2$ (i.e. $O(n^2m^2)$) comparisons.

Fig. 3 illustrates a trace for the Brute-Force algorithm with respect to the main matrix A: 6×6 and B: 2×2, which are presented in Fig. 2. The elements of the first five rows and those of the first five columns are inspected as potential upper-left corner matches. For various iterations, the shaded areas in the main matrix represent the elements which are compared with the corresponding ones of the submatrix. The total number of comparisons required to return MSM(A, B) = {A(1, 4), A(4, 5)} is 46 comparisons.

```
Algorithm Brute-Force Matrix   Submatching : MSM(A, B)
    // A: nXn and B: mXm, where m<=n
    MSM= {};
    mrow_idx = 1;  mcol_idx = 1; matched_elements = 0;
    while (mrow_idx <= n-m+1) do {
        match = UnKnown;
        matched_elements = 0;
        srow_idx = 1;  scol_idx = 1;
        candidate_r_idx = mrow_idx;   candidate_c_idx = mcol_idx;
        while ( (srow_idx <= m) && (match= =UnKnown) &&
                (A (mrow_idx, mcol_idx) = = B (srow_idx, scol_idx)) ) do
        {
            matched_elements = matched_elements + 1;
            if (matched_elements = = size(B))
                match = Found;
            else
            {   scol_idx = scol_idx+1;
                mcol_idx = mcol_idx+1;
                if  (scol_idx > m) {
                    srow_idx = srow_idx +1;
                    scol_idx = 1;
                    mrow_idx = mrow_idx +1;
                    mcol_idx = mcol_idx – m; }

                if  (mcol_idx > n)
                    srow_idx = m+1;
            } // end else.
        } // end while.
        mrow_idx = candidate_r_idx;  mcol_idx = candidate_c_idx;
        if (match = = Found)
            MSM ←A(candidate_r_idx, candidate_c_idx);
        mcol_idx = mcol_idx +1;
        if  (mcol_idx > n-m+1)
        {   mrow_idx = mrow_idx +1;
            mcol_idx = 1;
        } //  end if.
    } // end while.
Return MSM;
```

Fig. 1: Algorithm brute-force matrix submatching



0	0	0	0	1	1
1	1	1	1	1	1
1	1	0	0	1	1
0	0	1	1	0	1
0	0	0	0	1	1
1	1	1	0	0	0

0	1
1	1

Fig. 2: Example of main matrix A: 6×6 and submatrix B: 2×2

CHAIN CODE BASED METHOD

The matrix submatching or matrix containment problem implies searching for a pattern in the form of a matrix inside a larger matrix. The brute-force algorithm tends to work well for matrices which have no assumptions with respect to their contents. This section introduces another solution for the matrix submatching problem based on chain coding which is a succinct way of representing a list of points[6]. Only a starting point is represented by its location while the other points are represented by successive displacements from point to point along a certain path. For several applications of matrices such as image processing, a matrix tends to have repeating adjacent values representing objects. Although, the proposed solution works for general grey values of elements in matrices, the algorithm will be discussed with respect to binary matrices. Using the Chain-code based technique, the process of matrix submatching goes through two phases; namely, transformation and matching.

Transformation phase: "chain code matrix transformation" The objective of the transformation phase is to convert the main matrix and submatrix into two sets of vectors with each vector represents the chain code of the elements of the corresponding row in the original matrix. The chain code based transformation takes advantage of repeating values of

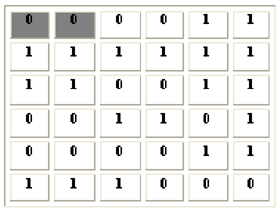
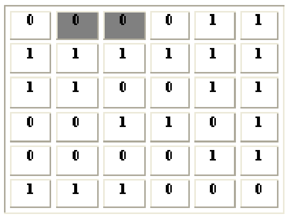
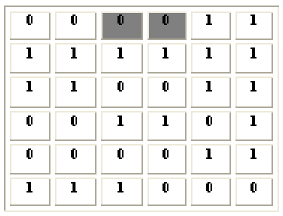
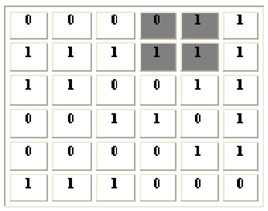
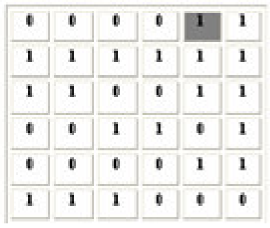
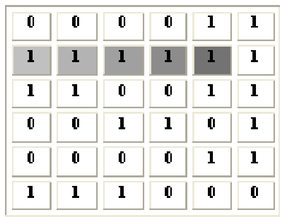
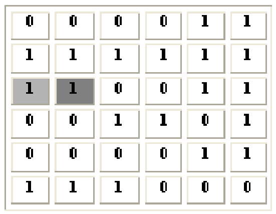
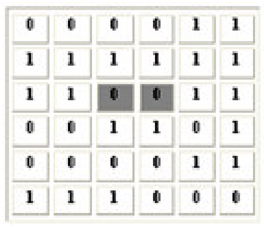
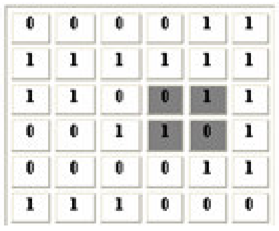
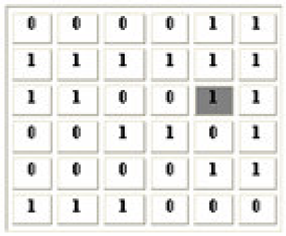
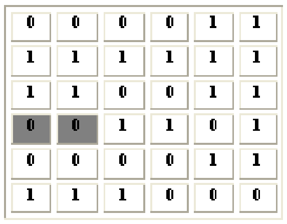
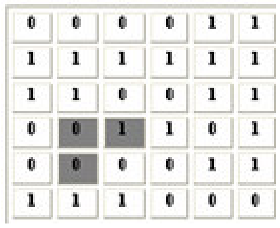
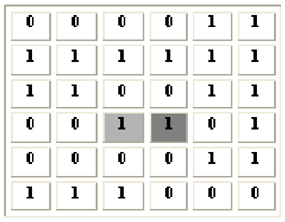
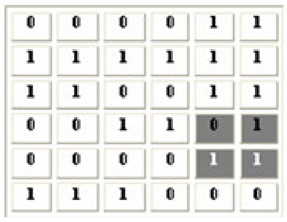
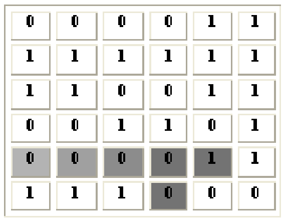
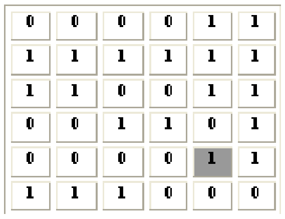
Row = 1  # of comparisons = 2 # of occurrences = 0 MSM(A,B) = { }	Row = 1  # of comparisons = 4 # of occurrences = 0 MSM(A,B) = { }	Row = 1  # of comparisons = 6 # of occurrences = 0 MSM(A,B) = { }	Row = 1  # of comparisons = 10 # of occurrences = 1 MSM(A,B)={A(1, 4) }
Row = 1  # of comparisons = 11 # of occurrences = 1 MSM = { A(1, 4) }	Row = 2  # of comparisons = 16 # of occurrences = 1 MSM = { A(1, 4) }	Row = 3  # of comparisons = 18 # of occurrences = 1 MSM = { A(1, 4) }	Row = 3  # of comparisons = 20 # of occurrences = 1 MSM = { A(1, 4) }
Row = 3  # of comparisons = 24 # of occurrences = 1 MSM = { A(1, 4) }	Row = 3  # of comparisons = 25 # of occurrences = 1 MSM = { A(1, 4) }	Row = 4  # of comparisons = 27 # of occurrences = 1 MSM = { A(1, 4) }	Row = 4  # of comparisons = 30 # of occurrences = 1 MSM = { A(1, 4) }
Row = 4  # of comparisons = 32 # of occurrences = 1 MSM = { A(1, 4) }	Row = 4  # of comparisons = 36 # of occurrences = 2 MSM = {A(1, 4), A(4, 5)}	Row = 5  # of comparisons =45 # of occurrences = 2 MSM = {A(1, 4), A(4, 5)}	Row = 5  # of comparisons =46 # of occurrences = 2 MSM = {A(1, 4), A(4, 5)}

Fig. 3: Trace for the brute-force algorithm with respect to matrices A and B in Fig. 2

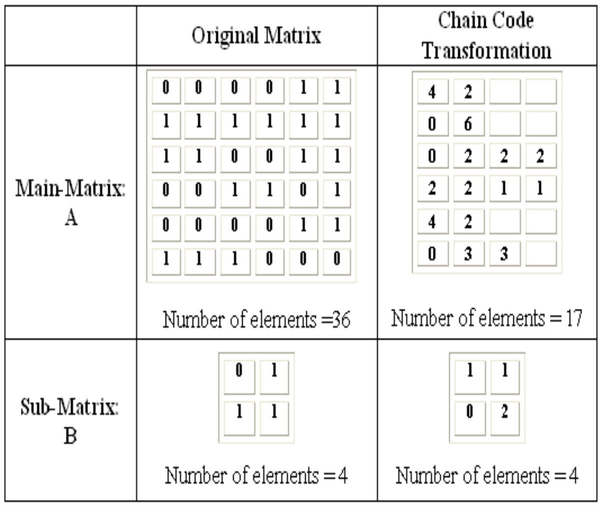
```
// $a'(i, 1)$ is the number of the first consecutive zero-value elements in $A_i$ starting with $a(i, 1)$.
  $a'(i, 1) \leftarrow 0$      if    $a(i, 1)=1$
  $a'(i, 1) \leftarrow r_1$    if    $a(i, 1)= a(i, 2)= \ldots = a(i, r_1)=0$, where $r_1 <= n$
  IF ($r_1 == n$) THEN { $k_i =1$; STOP}

// $a'(i, 2)$ is the number of the next consecutive one-value elements in $A_i$ starting with $a(i, r_1+1)$.
  $a'(i, 2) \leftarrow r_2$    if    $a(i, r_1+1)= a(i, r_1+2)= \ldots = a(i, r_1+ r_2)=1$, where $(r_1+ r_2) <= n$
  IF (($r_1+ r_2$)== n) THEN { $k_i =2$; STOP}

// $a'(i, 3)$ is the number of the next consecutive zero-value elements in $A_i$ starting with $a(i, r_1+ r2+1)$.
  $a'(i, 3) \leftarrow r_3$    if    $a(i, r_1+ r_2 +1)= a(i, r_1+ r_2+2)= \ldots = a(i, r_1+ r_2+ r_3)=1$, where $(r_1+ r_2+r_3) <= n$
  IF (($r_1+ r_2+r_3$)== n) THEN { $k_i =3$; STOP}
..., and so on.
```

Fig. 4: Chain code matrix transformation rules

successive elements within a specific row in order to reduce the size of the main and sub matrices. During the transformation (i.e. pre-processing) phase of matrix A, starting with the first row, each row $A_i[a(i, 1), a(i, 2), a(i, 3), \ldots, a(i, n)]$ in the nXn matrix is parsed and transformed into a vector $A_i'[a'(i, 1), a'(i, 2), a'(i, 3), \ldots, a'(i, k_i)]$, where $k_i \le n$ is the length of the vector A_i' corresponding to row A_i, i=1, 2, …, n. The contents of the vector A_i' will be determined as per Fig. 4.

It can be seen from the previous description that the first element of the vector A_i' represents the number of consecutive zeros starting with a(i, 1) of row A_i. However, if a(i, 1) contains one instead of zero, the first element of the A_i' vector will be assigned zero. The second element of the A_i' vector will be assigned the number of the next successive ones while the third element will receive the number of the next successive zeros and so on. All rows of the main matrix and those of the submatrix will be transformed in a similar fashion.

Figure 5 displays the chain code transformation for main matrix A: 6×6 and submatrix B: 2×2. Obviously, the transformation vectors corresponding to the rows of a particular matrix may not be of equal sizes. Actually, the size of the transformation vector A_i' corresponding to row A_i of n elements may become as small as one in the best case. For example, $A_i = [0, 0, 0, \ldots, 0]$ will be transformed into $A_i' = [n]$. However, when $A_i = [1, 0, 1, \ldots, 0/1]$, the transformation vector is $A_i' = [0, 1, 1, 1, \ldots, 1]$ and will have its maximum possible size; i.e. n+1. The size reduction of matrices using chain code transformation is more substantial when the matrix frequently contains continuous streams of identical values. This is typical in applications related to image processing, voice representation matrices and traffic control.



	Original Matrix	Chain Code Transformation
Main-Matrix: A	$\begin{array}{cccccc} 0&0&0&0&1&1\\ 1&1&1&1&1&1\\ 1&1&0&0&1&1\\ 0&0&1&1&0&1\\ 0&0&0&0&1&1\\ 1&1&1&0&0&0 \end{array}$ Number of elements =36	$\begin{array}{cccc} 4&2&&\\ 0&6&&\\ 0&2&2&2\\ 2&2&1&1\\ 4&2&&\\ 0&3&3& \end{array}$ Number of elements = 17
Sub-Matrix: B	$\begin{array}{cc} 0&1\\ 1&1 \end{array}$ Number of elements = 4	$\begin{array}{cc} 1&1\\ 0&2 \end{array}$ Number of elements = 4

Fig. 5: Chain code transformation for matrices A and B of Fig. 2

We can notice from Fig. 5 that the transformation phase reduces the size of the matrices depending on sequential repetition of the values in the matrix. This reduction in size will decrease the time of matrix searching using our proposed algorithm comparing with the brute-force algorithm that works on the original matrices.

Table 1: Variables utilized in the chain code based search Algorithm

Variable name	Description
TMM	n X n transformed main matrix corresponding to main matrix A.
TSM	m X m transformed sub matrix corresponding to submatrix B.
Offset_row	Indicates the offset of the row in the original main matrix A.
Offset_col	Indicates the offset of the column in the original main matrix A.
Result	A 2-D matrix [row, col], where row is Offset_row and col is Offset_col.
Flag	Boolean variable, set to "True" if the start point of matrix submatching is found.
Stop	Boolean variable which indicates the end of the matching process; i.e. when the end of TMM is reached.
Break	Boolean variable which indicates the end of the matching process; i.e. when the end of TSM is reached.
Sub_col	Number of columns in the current Sub_row.
S_col	Number of columns in the original submatrix B.
Main_col	Number of columns in the current Main_row.
M_col	Number of columns in the original matrix A.
i, j	Row and column counters in TMM, respectively.
n, m	Row and column counters in TSM, respectively.

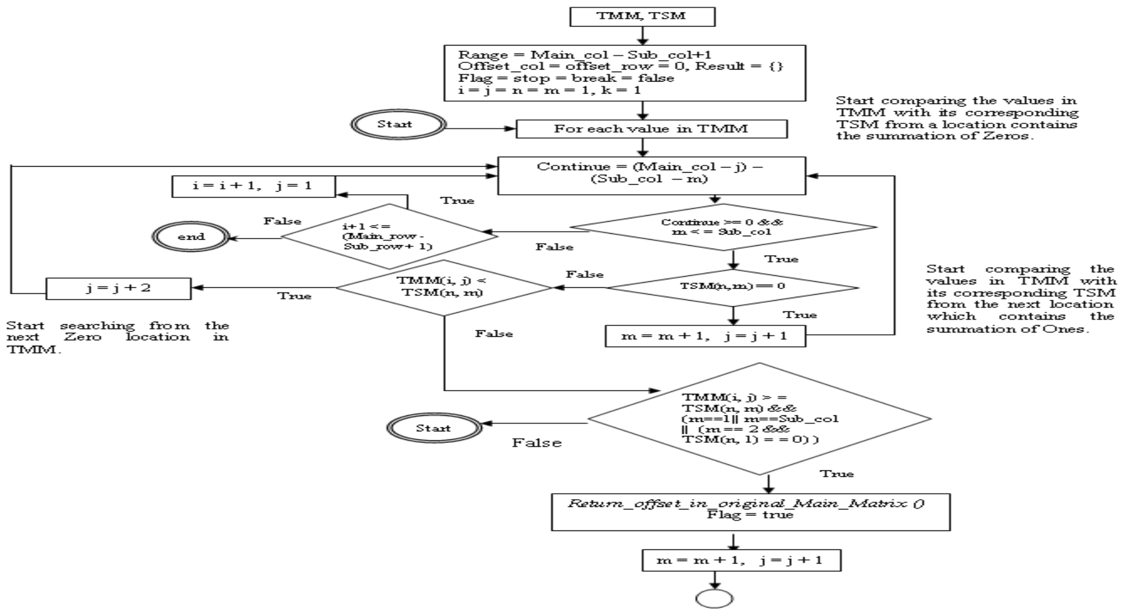

TMM, TSM
Range = Main_col – Sub_col+1
Offset_col = offset_row = 0, Result = {}
Flag = stop = break = false
i = j = n = m = 1, k = 1
Start comparing the values in TMM with its corresponding TSM from a location contains the summation of Zeros.
Start
For each value in TMM
Continue = (Main_col – j) – (Sub_col – m)
i = i + 1, j = 1
True
False
i+1 <= (Main_row - Sub_row+ 1)
end
Continue >= 0 && m < = Sub_col
False
True
Start comparing the values in TMM with its corresponding TSM from the next location which contains the summation of Ones.
j = j + 2
True
TMM(i, j) < TSM(n, m)
False
TSM(n,m) == 0
True
Start searching from the next Zero location in TMM.
False
m = m + 1, j = j + 1
TMM(i, j) > = TSM(n, m) && (m==1 || m==Sub_col || (m == 2 && TSM(n, 1) = = 0))
Start
False
True
Return_offset_in_original_Main_Matrix ()
Flag = true
m = m + 1, j = j + 1

Fig. 6: Flow chart for finding the first point of match in TMM

Search phase: "matrix submatching algorithm" The task of a submatching algorithm is to find all occurrences of a two-dimensional matrix B: m×m in a two-dimensional matrix A: n×n. This section introduces a matrix submatching algorithm which utilizes the chain code transformation vectors of A and B. Table 1 states the variables used in the search phase while Fig. 6 illustrates the first part of the algorithm.

The chain code based search algorithm builds on the assumption that each vector of the transformed matrices starts with the count of zeros. Obviously, if the first value in the vector is zero, it reflects that the corresponding row in the original matrix starts with one. Fig. 6 illustrates a flow chart for finding the first point of match in TMM.

If TMM starts with a number of zeros or ones larger than that in TSM, we call the function: Check leading zeros or ones as explained in Fig. 7 to search for submatrix matching sequentially.

If the start point of match is found, the function: Return offset in original Main Matrix () as described in

```
Algorithm: Check leading zeros or  ones()

    1.  if (TMM (i, j) > TSM (n, m))
    2.      if ( TSM(n,1) = =S_col  || [ (TSM (n, 1) = = 0 && TSM (n, 2) = =S_col )])
    3.              Leading Zeros or ones in the  TMM , start searching sequentially like the brute force
        algorithm.
    4.  continue;
```

Fig. 7: Check leading zeros or ones function

```
Algorithm: Return offset in original Main Matrix ()
   // find the exact row and column in the original Main Matrix.
    1.  counter = 0;
    2.  if (Flag  = = false)
    3.        next = j; // to calculate the next position from which we will continue searching.

    4.  if (j > 1)
    5.  {     for (d = 1 : j - 1)
    6.              counter = counter + TMM  (i, d); } // find the summation of previous values
    7.  else
    8.        counter = 0;

    9.  offset_row = i;
    10. offset_col =  TMM  (i, j) –   TSM  (n, m) + counter + 1;    // we add 1 because the matrix's
        index starts from 1.

    11. Return offset_row, offset_col;
```

Fig. 8: Return offset in original main matrix function

Fig. 8 is invoked to find the row and column offsets in the original main matrix. Then, Flag is set to True.

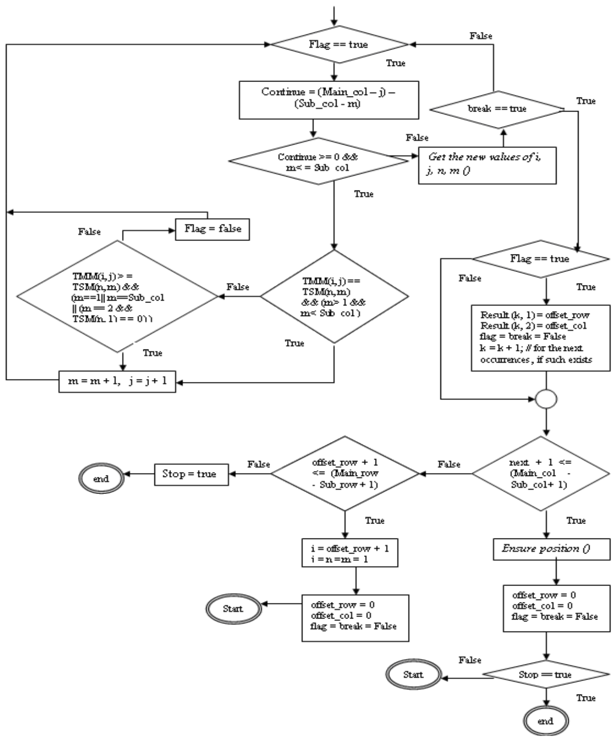
After finding the first point of match, we continue searching for potential other points of match as per Fig. 9. Search is terminated when one of the following two cases occurs:

- If the last element of TSM Matrix has been reached, then a sub-matrix match has been found
- If the value of TMM < the corresponding in TSM, then Flag is set to FALSE

If the end of the current row in TSM has been reached, the function: Get the new values(i, j, n, m) will be called in order to update the values of counters i, j, n, and m. Fig. 10 shows how the function works.

To update the value of counter j, function Return offset in TMM () as demonstrated in Fig. 11 will be invoked. This function will return the exact column in the next row in TMM to start search.

After finding the value of j (i.e., lines 1-5) of Fig. 11 which indicates the column in the next row in TMM, we continue searching while maintaining that m (i.e., column counter in TSM) is pointing to a valid position. If the row starts with 1, then the first column will contain 0. In this case, we increment m to point to the next location (i.e., line 6). Then, we compare the location to which j is pointing with the corresponding



Flag == true
True
False
Continue = (Main_col – j) – (Sub_col - m)
break == true
True
False
Continue >= 0 && m< = Sub_col
Get the new values of i, j, n, m ()
True
Flag = false
False
TMM(i, j) > = TSM(n,m) && (m==1|| m==Sub_col || (m == 2 && TSM(n, 1) == 0))
False
TMM(i, j) == TSM(n,m) && (m> 1 && m< Sub_col)
True
True
m = m + 1, j = j + 1
Flag == true
False
True
Result (k, 1) = offset_row
Result (k, 2) = offset_col
flag = break = False
k = k + 1; // for the next occurrences , if such exists
end
Stop = true
False
offset_row + 1 <= (Main_row - Sub_row + 1)
False
next + 1 <= (Main_col - Sub_col+ 1)
True
True
i = offset_row + 1
i = n = m = 1
Ensure position ()
Start
offset_row = 0
offset_col = 0
flag = break = False
offset_row = 0
offset_col = 0
flag = break = False
Start
False
Stop == true
True
end

Fig. 9: Flow chart for finding subsequent points of match after detecting the first one

```
Algorithm: get the new values( i , j, n, m)
  // check if we reach the end of the current row in the TSM.
    1.  if (m > Sub_col
    2.  {  m = 1;
    3.     if (n+1 > Sub_row) {
    4.           n = 1;   break = true; // to break from the while. Return ;}

    5.     else{ n = n + 1;
    6.           if (i + 1 <= Main_row)
    7.               i = i + 1;
    8.           else {  stop = true;  break = true; // to break from the while. Return ;}
    9.          } // end else.

    10.         [i, j] = Return offset in TMM ()
    11.  } // end if.

    12.  else {  m = m + 1;  j = j + 1; }
    13. Return i, j, n, m;
```

Fig. 10: Get the new values of i, j, n, m function

```
Algorithm: Return offset in TMM ()
  // return the exact column in the next row in TMM to start searching from.
    1.  w = 0;   sum = 0;
    2.  while (sum < offset_col) {
    3.          w = w + 1;
    4.          sum = sum + TMM (i, w); }
    5.  j = w;

    6.  if (TSM(n, m) = = 0) m = m + 1;

    7.  if (mod (j, 2) ~ = 0 && mod (m, 2) = = 0) {flag = false; break = true; Return; } // j pointing
        to a one location, while the search must start from a zero location.

    8.  if (mod (j, 2) = = 0 && mod (m, 2) ~ = 0)      {flag = false; break = true;    Return; } // j
        pointing to a zero location, while the search must start from a one location.

    9.  [w, x] = Return offset in original Main Matrix ();

    10.  if (~ (x >= offset_col && x <= offset_col + (TSM (n, m) - 1) ) )
    11.    {flag = false; break = true; Return; }
```

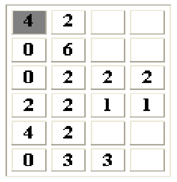
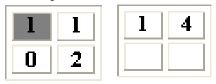
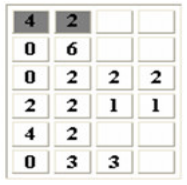
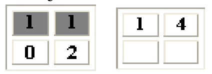
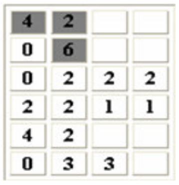
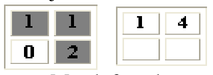
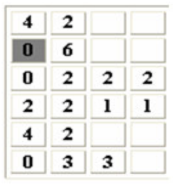
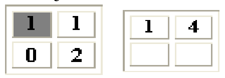
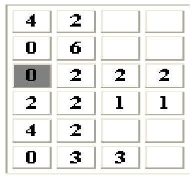
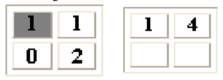
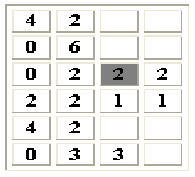
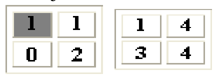
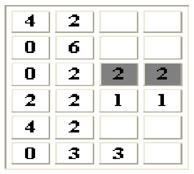
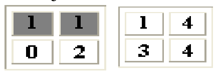
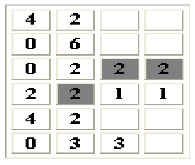
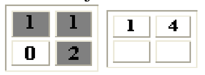
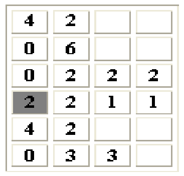
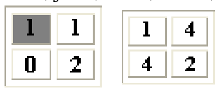
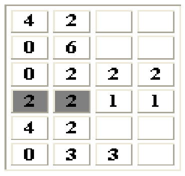
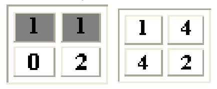
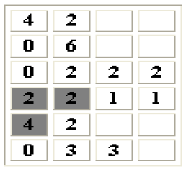
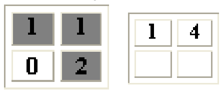
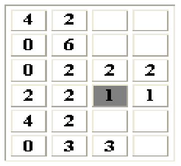
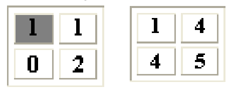
Fig. 11: Return offset in TMM function

```
Algorithm: Ensure position ( )

 // ensure that the searching process will start from a location representing 0.

    1.  if (mod (( next + 1), 2) ~ = 0)
    2.       j = next + 1;
    3.  else
    4.     { j = next + 2; }
    5.  m = 1;  n = 1;  i = offset_row ;
    6.  Return i, j, n, m;
```

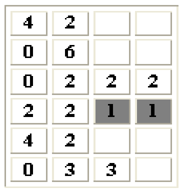
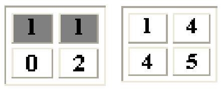
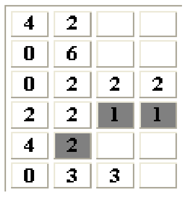
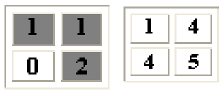
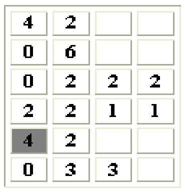
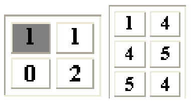
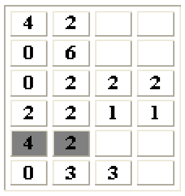
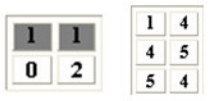
Fig. 12: Ensure position function

one in TSM. If the two are not pointing to the same location, we set Flag to false. If they are pointing to the same location, the function Return offset in original Main Matrix () will be called to check that the value of j remains in the correct boundaries of search (i.e., lines 7-11). Then, we check if the flag is true to register the offset_row and offset_col in result matrix as the first occurrence. Fig. 12 displays the function which validates location correctness. The whole search process will stop once we reach the end of TMM.

Fig. 13 shows a trace using the chain code based algorithm for the main and sub matrices shown in Fig. 5. While the brute-force algorithm requires 46 comparisons to complete the search of the indicated matrices, the chain-code based algorithms requires only 17 comparisons to find all occurrences. This is due to the reduction in size caused by the transformation phase by almost 50%. A comprehensive experimental comparison between the two algorithms in terms of the required number of comparisons to find all occurrences is discussed in the following section.

No. of comparisons: 1  TMM (i, j) >= TSM (n, m) offset_row = 1 offset_col =(4 - 1)+ 0+1 = 4 i = 1, j = 1, n = 1, m = 1, 	No. of comparisons:2  TMM (i, j) >= TSM (n, m) offset_row = 1 offset_col = 4 i = 1, j = 2, n = 1, m = 2, 	No. of comparisons:3  TMM (i, j) >= TSM (n, m) offset_row = 1 offset_col = 4 i = 2, j = 2, n = 2, m = 2.  Match found	No. of comparisons: 4  TMM (i, j) < TSM (n, m) offset_row = 0 offset_col = 0 i = 2, j = 1, n = 1, m = 1, 
No. of comparisons: 5  TMM (i, j) < TSM (n, m) offset_row = 0 offset_col = 0 i = 3, j = 1, n = 1, m = 1, 	No. of comparisons:6  TMM (i, j) >= TSM (n, m) offset_row = 3 offset_col = (2 - 1)+ 2+1 = 4 i = 3, j = 3, n = 1, m = 1, 	No. of comparisons:7  TMM (i, j) >= TSM (n, m) offset_row = 3 offset_col = 4 i = 3, j = 4, n = 1, m = 2. 	No. of comparisons:8  TMM (i, j) >= TSM (n, m) offset_row = 0 offset_col = 0 i = 4, j = 2, n = 2, m = 2.  In TMM start searching from j = 2, incorrect lower boundary.
No. of comparisons:9  TMM (i, j) >= TSM (n, m) offset_row = 4 offset_col = (2 - 1)+0+1 = 2 i = 4, j = 1, n = 1, m = 1, 	No. of comparisons:10  TMM (i, j) >= TSM (n, m) offset_row = 4 offset_col = 2 i = 4, j = 2, n = 1, m = 2 	No. of comparisons:11  TMM (i, j) >= TSM (n, m) offset_row =0 offset_col = 0 i = 5, j = 1, n = 2, m = 2.  In TMM start searching from j = 1, which indicate a location which contains summation of consecutive zero.	No. ofcomparisons:12  TMM (i, j) >= TSM (n, m) offset_row = 4 offset_col =(1 - 1)+4+1 = 5 i = 4, j = 3, n = 1, m = 1. 

No. of comparisons:13	No. of comparisons:14	No. of comparisons:15	No. of comparisons:16


TMM (i, j) >= TSM (n, m)
offset_row = 4
offset_col = 5
i = 4, j = 4, n = 1, m = 2,

	
	TMM (i, j) >= TSM (n, m)
	offset_row = 4
	offset_col = 5
	i = 5, j = 2, n = 2, m = 2
	
	Match found
		
		TMM (i, j) >= TSM (n, m)
		offset_row =5
		offset_col = (4-1)+0+1= 4
		i = 5, j = 1, n = 1, m = 1.
		
			
			TMM (i, j) >= TSM (n, m)
			offset_row = 5
			offset_col = 4
			i = 5, j = 2, n = 1, m = 2.
			

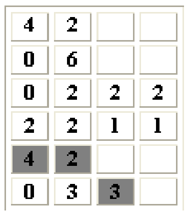
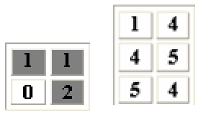
No. of comparisons:17  TMM (i, j) >= TSM (n, m) offset_row = 5 offset_col = 4 i = 6, j = 3, n = 2, m = 2.  In TMM start searching from j = 3, which indicate a location contains summation of consecutive zero.			

Fig. 13: Trace for the chain-code based matrix submatch algorithm for matrices A and B in Fig. 5.

RXPERIMENTAL RESULTS

The brute-force and chain-code based algorithms are considered sequential search mechanisms for the matrix submatching problem. In order to experimentally compare the performance of both algorithms, we randomly generated a database for main matrices with sizes 50×50, 75×75, 100×100 and 200×200 and another one for submatrices with sizes 10×10, 15×15, 25×25, 30×30, 35×35, 40×40 and 45×45 using Matlab. The databases contain 1000 occurrences of each indicated size and the average numbers of comparisons required by both algorithms to find the occurrences of submatricies in the corresponding main matrices were computed. The outcome of the experiments is summarized in Fig. 14. Our experiments clearly show that the chain code based algorithm requires half the number of comparisons required by the brute-force approach. This is basically attributed to the compression in size due to the preprocessing phase of the chain-code approach. For several applications, it is typical that a database of matrices exists and a query is posed against the database to retrieve all matrices which contain an incoming sub matrix[7, 11]. In such cases, the preprocessing phase for the main matrices needs to be done only once.

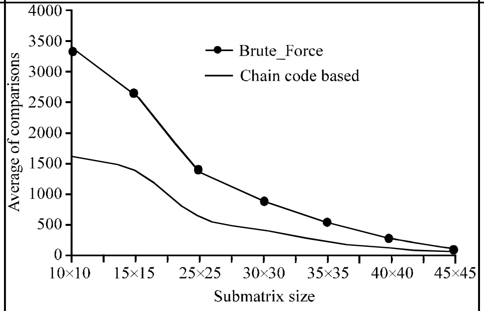
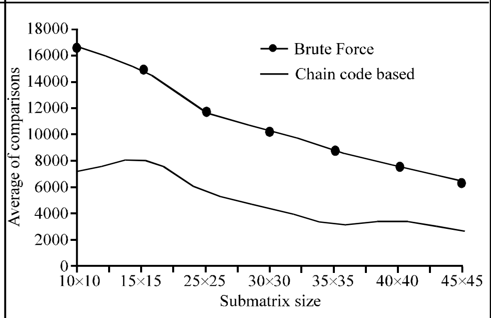
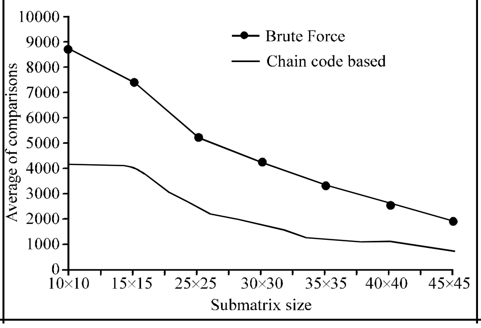
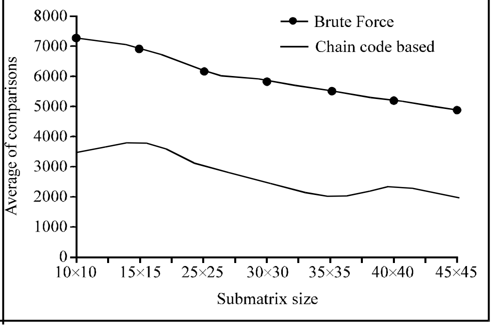
Main matrix size	No. of comparison required by Brute Force Vs. chain-code based algorithm	Main matrix size	No. of comparison required by Brute Force Vs. chain-code based algorithm
50×50		100×100	
75×75		200×200	

Fig. 14: No of Comparisons required by the brute-force and chain-code based Algorithms

Table 2: Average percentage of square matrix size (NXN) reduction due to preprocessing phase

Matrix N=	Percentage of size reduction	Matrix N=	Percentage of size reduction
2	0	500	49.8
5	30	1,000	49.9
10	40	5000	50.0
15	43.3	10000	50.0
20	45.0	25000	50.0
25	46.0	50000	50.0
50	48.0	100000	50.0
75	48.7	500000	50.0
100	49.0	1000000	50.0

Table 2 demonstrates the average percentage of size reduction for randomly generated square binary matrices with various sizes. The maximum average percentage of size reduction is 50%.

CONCLUSION

This article brings focus to the matrix submatching operation as an essential problem to be solved for many applications including watermarking, geographic information systems and pattern recognition. Most of these applications start with a database of matrices and require the retrieval of those matrices which contain an incoming matrix. The chain code based approach presented in this paper consists of two phases; namely, transformation and matching. The transformation phase reduces the sizes of all relevant matrices by nearly half of their original sizes bringing about clear saving in the number of comparisons when compared with the brute force approach. Although, this paper demonstrated superiority of the chain-code approach for binary square matrices, the results hold true for general matrices.

ACKNOWLEDGMENT

The authors would like to acknowledge the contribution of Thaer Al-Ibaisi and Khitam Jbara in obtaining experimental results.

REFERENCES

1. Angiulli, F. and E. Cesario, 2006. A Greddy Search Approach to Co-clustering Sparse Binary Matrix, 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06), pp: 363-370.
2. Bronson, R., 1989. Schaum's Outline of Theory and Problems of Matrix Operations, McGraw-Hill.

3. Coppersmith, D. and S. Winograd, 1990. Matrix multiplication via arithmetic progressions, J. Symbolic Computat., 9: 251-280.
4. Cormen, T.H., C.E. Leiserson, R.L. Rivest, and C. Stein, 2001. Introduction to Algorithms. 2^{nd} Edn., The MIT Press.
5. Jane, A., 1995. Parallel search with matrices with sorted column, 7th IEEE Symposium on Parallel and Distributed Processing, pp: 224-228.
6. Mei-Chen, Y., Y. Huang and J. Wang, 2002. Scalable Ideal-Segmented Chain Coding, IEEE international conference on image processing.
7. Knuth, D.E., J.H. Morris and V.R. Pratt, 1977. Fast pattern matching in strings, SIAM. J. Comput., 6 (2): 323-350.
8. Koyuturk, M. and A. Grama, 2006. Nonorthogonal decomposition of binary matrices for bounded-error data compression and analysis, ACM transactions on mathematical software, 32 (1): 33-69.
9. Robinson, S., 2005. Toward an optimal algorithm for matrix multiplication. SIAM News, 38 (9).
10. Shen, X. and Q. Hu, 1992. Approximate submatrix matching problems. Proceeding of the ACM Symposium on Applied Computing (SAC'92), pp: 993-999.
11. Sleit, A., W. AlMobaideen, A. Baarah and A. Abusitta, 2007. An efficient pattern matching algorithm. J. Applied Sci., 7 (18): 2691-2695.
12. Smith, P., 1991. Experiments with a very fast substring search algorithm. Software-practice and experience. 21 (10): 1065-1074.