

Transformations on Images Stored in Peano Mask Trees

Mohammad Kabir Hossain,
Department of Computer
Science and Engineering,
North South University,
Dhaka-1213, Bangladesh
mkhossain@northsouth.ed,

Shams Mahmood Imam
Department of Computer
Science and Engineering,
North South University,
Dhaka-1213, Bangladesh
shams_imam@hotmail.com

S. M. Rezaul Hoque
Department of Computer
Science and Engineering,
North South University,
Dhaka-1213, Bangladesh
rezahok@hotmail.com

Abstract

Peano Trees are a data-mining ready data structure that can also be used for image compression. The ability to be able to rotate, scale or translate images is an integral part of image processing while searching through images in the image database. This paper presents algorithms for such transformations of images in the two dimensional plane. The paper assumes that the images to be worked upon are stored the Peano Mask Tree format (Peano Trees are introduced below). The algorithm will provide the advantage of image processing while the image is in the Peano Tree format, it will not be necessary to generate the original bit pattern of the image to perform any of the transformations.

1. Introduction

There are a wide variety of applications where pattern images are required to be aligned over a reference image so that common pixels of the images are present in the same location. This process is useful in the alignment of an acquired image over a template, a time series of images of the same scene, or the separate bands of a composite image (coregistration). Two practical applications of this process are the alignment of radiology images in medical imaging and alignment of satellite images for environmental study [1].

Changing the orientation or parameters of the imaging sensor can cause differences of alignment. Hence, a translation in the position of the sensor gives rise to translated images. To undo this translation effect we require translating the generated image in the opposite direction. In this paper, we present such an algorithm to translate images stored in the compressed

Peano Mask tree format. Along with translations, rotations and scaling operations need also be performed. We also present algorithms for these transformations.

2. Introduction to Peano Trees

2.1. The Peano Count Tree (P-Tree¹)

P-trees are a lossless, compressed, and data-mining-ready data structure. This data structure has been successfully applied in data mining applications ranging from Classification and Clustering with K-Nearest -Neighbor, to Classification with Decision Tree Induction, to Association Rule Mining [3,4,5,6].

The Peano Count Tree (P-tree) is a quadrant-based lossless tree representation of the original spatial data [5,6]. The general concepts of P-trees are to recursively divide the entire spatial data into quadrants and record the count of 1-bits for each quadrant, thus forming a quadrant count tree. Using P-tree structure, all the count information can be calculated quickly. This facilitates efficient ways for data mining.

For example, a P-Tree for the given 8 x 8 image of single bits, which we also call the data bit array, is explained below:

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

Figure 1. 8-bit by 8-bit image

¹ Patents are pending for P-Tree technology

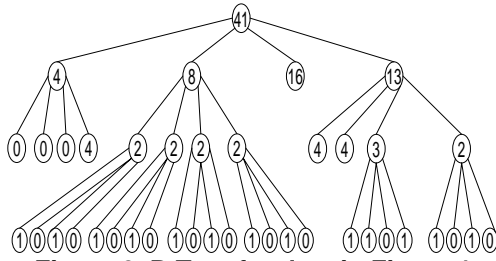


Figure 2. P-Tree for data in Figure 1

In this example, 41 is the number of 1's in the entire image. This root level is labeled level 0. The numbers 4, 8, 16, and 13 found at the next level (level 1) are the 1-bit counts for the four major quadrants in raster order, or Z order (upper left, upper right, lower left, and lower right). This process of storing the count of the number of 1's for each of the major quadrants is repeated. When quadrants are composed entirely of 1-bits (called pure-1 quadrants), sub-trees are not needed, and these branches terminate. Similarly, quadrants composed entirely of 0-bits are called pure-0 quadrants, which also cause termination. This pattern is continued recursively using the Peano, or Z-ordering (recursive raster ordering), of the four sub-quadrants at each new level. Eventually, every branch terminates (since, at the "leaf" level, all quadrants are pure).

The P-tree structure can be viewed as a data-mining-ready structure as it facilitates efficient ways for data mining. P-trees have the following features:

- P-trees contain 1-count for every quadrant of every dimension.
- The P-tree for any sub-quadrant at any level is simply the sub-tree rooted at that sub-quadrant.
- A P-tree leaf sequence (depth-first) is a partial run length compressed version of the original bit-band.
- Basic P-trees can be combined to reproduce the original data (P-trees are lossless representations).
- P-trees can be partially combined to produce upper and lower bounds on all quadrant counts.
- P-trees can be used to smooth data by bottom-up quadrant purification (bottom-up replacement of mixed counts with their closest pure counts)..

2.2. Peano Mask Tree (PM Tree)

A variation of the P-tree data structure, the Peano Mask Tree (PM-tree), is a similar structure in which masks, rather than counts, are used. In a PM-tree, we use three-value logic to represent pure-1, pure-0, and mixed quadrants. (A 1 denotes pure-1; 0 denotes pure-0; and m denotes mixed.) The PM-tree for the previous example is also given in Figure 3. We can easily

construct the original P-tree from its PM-tree by calculating the counts from the leaves to the root in a bottom-up fashion. A PM-tree is just an alternative implementation for a Peano Count tree [2].

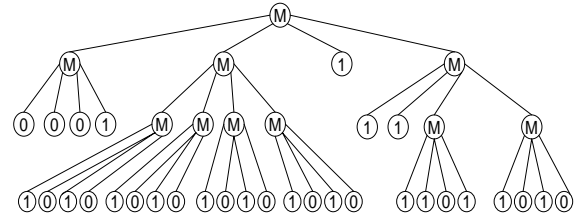


Figure 3. PM-Tree for data in Figure 1.

It is worth describing here the data structure used to implement a Peano Mask tree. The Peano Mask tree can be represented by a linked data structure in which each node is an object. Each node contains a *type* field, which represents what type of node it is (0, 1 or M) and a *position* field storing which child it is of its parent (1, 2, 3 or 4). In addition, each node contains fields *parent* and an array of node *child*[4] that point to the nodes corresponding to its parent and children from left to right, respectively. If a child or parent is missing, the appropriate field contains the value NIL. The root node is the only node in the Peano Mask tree, which has its parent field as NIL. The child nodes are numbered from 1 to 4 according to their array indices, which set their position values.

3. Transformation Operations

3.1. The Translation Algorithm

It can be noted that any translation may be represented as a vector comprising of two components. One component can be used to represent a right or left shift of the pixels of the image while the other component can be used to represent a up or down shift. Hence, any translation consisting of integer components can be achieved via a sequence of right/left shifts followed by a sequence of up/down shifts. We present here algorithms for left and down shifts. The corresponding algorithms for right-shifts and up-shifts are symmetric and are not presented here. We assume that the image dimensions are equal powers of 2, say 2^M .

3.1.1. The Circular Left-Shift. We present here the algorithm to carry out a circular left shift operation of 2^n where $n < M$. We observe that the shift operations result in a change of the child nodes of the PM Tree at Level $(M-n)$. Let C_{xy} represent the nodes of the PM

Tree at Level (M-n-1). Hence, we need to devise a method to obtain the configuration for the new children of C_{xy} . We present below a method for the new ordering of the child nodes of a node C_{xy} . As can be seen from CIRCULAR-LEFT-SHIFT-NODE () the new arrangements of child nodes are that the children at position 2 and 4 become children at position 1 and 3, respectively. The children at position 1 and 3 of the immediate right neighbor of the current node become children at position 2 and 4, respectively. Before we can perform the shift operation on the PM Tree T we require expanding the PM Tree till Level M - n. This is achieved by recursively breaking down all Pure-1 and Pure-0 nodes of T till they are at level M - n. This is achieved via the call to function EXPANDTREE ().

Table 1. Circular left shift algorithm

```

CIRCULAR-LEFT-SHIFT-TREE (PMTREE T, SHIFT-SIZE 2^n)
1   EXPANDTREE(T, M - n)
2   for each NODE n at Level ( M-n-1 )
3     CIRCULAR-LEFT-SHIFT-NODE( n )
4   COLLAPSETREE(T)

CIRCULAR-LEFT-SHIFT-NODE ( NODE Cxy )
1   newCxy.child[ 1 ] ← oldCxy.child[ 2 ]
2   newCxy.child[2] ← IMMEDIATE-RIGHT-NEIGHBOR(
oldCxy ).child[ 1 ]
3   newCxy.child[ 3 ] ← oldCxy.child[ 4 ]
4   newCxy.child[ 4 ] ← IMMEDIATE-RIGHT-NEIGHBOR(
oldCxy ).child[ 3 ]

IMMEDIATE-RIGHT-NEIGHBOR( NODE C )
1   STACK S
2   while ( C.position ≠ 1 OR C.position ≠ 3 )
      AND C.parent ≠ NIL
3     PUSH( S, C.position )
4     C ← C.parent
5   if C.position = 1
6     C ← C.parent.child[ 2 ]
7   else if C.position = 3
8     C ← C.parent.child[ 4 ]
9   while S ≠ ∅
10    pos ← POP( S )
11    if pos = 4
12      newPos = 3
13    else
14      newPos = 1
15    C ← C.child[ newPos ]
16  return C

```

Below is an example of circular left shift of size 2 (i.e. 2^1) on the data of Figure 1 and PM Tree of Figure 3. First, we present a figure of the expanded tree if we wish to perform a shift of 2 (2^1) bits.

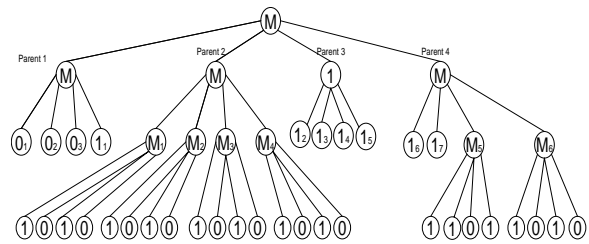


Figure 4. Expanded tree of Figure 3.

Next, we show a figure to show the path taken by the IMMEDIATE-RIGHT-NEIGHBOR() for a particular node.

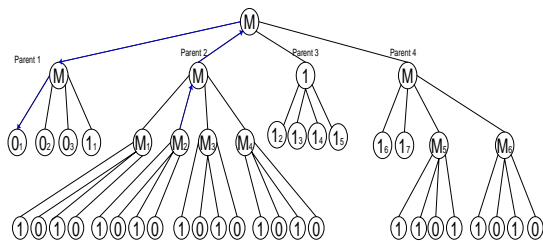


Figure 5. Example Path followed for NextNeighborRight()

After the call to CIRCULAR-LEFT-SHIFT-TREE (PMTREE T, SHIFT-SIZE 2^1), we get the following PM Tree:

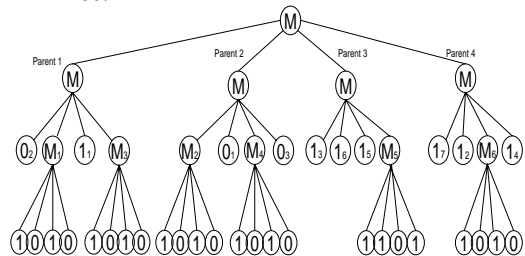


Figure 6. Circular shift left Tree of 2 units

The data array when regenerated from the PM Tree of figure 6 is:

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

Figure 7. Circular shift left by 2 units

which is exactly the data array, which we would have obtained if we had shifted the data bits in Figure 1 by 2 units.

3.1.2. The Circular Down-Shift. The circular down shift is similar to circular left shift but there are differences in the order of rearrangement of child nodes as well as the neighbor function that is called. As in circular left shift, we need to expand the PM Tree till Level $M - n$.

Table 2. Circular down shift's neighbor algorithm

```

IMMEDIATE-TOP-NEIGHBOR( NODE C )
1   STACK S
2   while ( C.position ≠ 3 OR C.position ≠ 4 )
      AND C.parent ≠ NIL
3     PUSH( S , C.position )
4     C ← C.parent
5   if C.position = 3
6     C ← C.parent.child[ 1 ]
7   else if C.position = 4
8     C ← C.parent.child[ 2 ]
9   while S ≠ ∅
10    pos ← POP( S )
11    if pos = 1
12      newPos = 3
13    else
14      newPos = 4
15    C ← C.child[ newPos ]
16  return C

```

3.1.3 Analysis of the Shift Algorithms. The running time of the algorithms is measured using the two parameters M , where 2^M is the one of the dimensions of the input binary data, and n , where 2^n is the size of the bits that requires to be shifted. These are the only two parameters we will use to define the time complexity of the algorithms mentioned above.

The sum of the nodes of a PM Tree up to and including the Level $M - n - 2$ is lower than the number of nodes at Level $M - n - 1$. Hence, the EXPANDTREE () takes at most $O(4^{(M-n-2)})$ running time. Since the for loops at lines 2 of both the CIRCULAR-LEFT-SHIFT-TREE() and CIRCULAR-DOWN-SHIFT-TREE() are entered $4^{(M-n-1)}$ times, this part of the code produces the leading term for the running time of the algorithm. In each of these loops, the IMMEDIATE-NEIGHBOR() methods are the only steps that require more than $O(1)$ time. In particular each call to IMMEDIATE-NEIGHBOR() requires $O(M)$ time since we are moving up the tree in each of the loops, and the height of the tree is M . Hence the overall running time of any of the SHIFT-TREE() methods is $O(M 4^{(M-n-1)}) = O(2^{\lg M + 2(M-n-1)})$. Note that this is an upper bound on the running time. A tighter bound on the running time can be achieved using amore detailed analysis.

The running time of the conventional shifting of a data array of size 2^M by 2^n units is $\Theta(2^M 2^n) = \Theta(2^{M+n})$

), since we require to translate a total of 2^{M+n} bits. For the mentioned shifting algorithm to perform better than this method;

$$\begin{aligned}
& 2^{\lg M + 2(M-n-1)} < 2^{M+n} \\
\Rightarrow & \lg M + 2M - 2n - 2 < M + n \\
\Rightarrow & \lg M + M < 3n + 2 \dots \dots \dots (i)
\end{aligned}$$

Whenever eq(i) is satisfied, the suggested algorithm will perform better than the conventional technique to shift any data array.

3.2. The Scaling Algorithm

The scaling operation is used to increase or decrease the dimensions of an image. In our algorithm we are only concerned with scaling operations that scale the image by a factor of two either to enlarge or to shrink the image. The key idea here is that during either operation the basic P-Tree structure remains unchanged. Hence only the dimensions of the image need to be changed, while minimal changes occur to the Peano Tree structure.

3.2.1. Scale-Up. The Scale-Up operation is the easier of the two scaling operations. This requires no changes to the P Tree structure. It only requires changing the dimensions of the image.

3.2.2. Scale-Down. The Scale-Down operation requires a little more observation. During the Scale-Down operation the image dimensions get smaller, hence the P Tree structure of the image also changes. It is observed that for a scale factor of 2^x we need to chop off the nodes at the lowermost x levels of the P Tree. We have noticed earlier that the leaves of any P Tree are always pure nodes. Hence after the chop-off we require to replace any existing M -nodes at the leaf levels with either '0' or '1'. A simple method is followed for this replacement. The 0:1 bit ratio is calculated for the binary data that can be generated using the M -node as the root. If this ratio is greater than or equal to 0.5 we replace the M by a 1, else we replace it by a 0. After a change is made, we need to move up the tree and see if it is possible to further collapse the tree.

Table 3. Scale down algorithm

```

SCALE-DOWN (NODE C)
1   // Make Changes to Image Dimensions
2   if C is a node in the level x from the bottom of the tree
3     if C.type = M
4       // Get the bit ratio
5       bit_ratio ← GET-BIT-RATIO (NODE C)
6       if bit_ratio >= 0.5
7         C.type = 1

```

```

8     else
9         C.type = 0
10    // Set all the child attributes to null.
11    C.child[ 0 ] ← null
12    C.child[ 1 ] ← null
13    C.child[ 2 ] ← null
14    C.child[ 3 ] ← null
15    // Handle changes that occur in the structure of the tree.
16    if C.parent ≠ null
17        UPDATE-NODE-TYPE ( C.parent )

```

```

UPDATE-NODE-TYPE (NODE C)
1    if C.type = M
2        t ← C.child[ 0 ].type
3        if t = C.child[ 1 ].type AND t = C.child[ 2 ].type
4            AND t = C.child[ 3 ].type
5            C.type ← t
6            if C.parent ≠ null
7                UPDATE-NODE-TYPE ( C.parent )

```

```

GET-BIT-RATIO (NODE C)
1    if C.child[ 0 ].type = M
2        ratio1 ← GET-BIT-RATIO ( C.child[ 0 ] )
3    else
4        ratio1 ← C.child[ 0 ].type
5    if C.child[ 1 ].type = M
6        ratio2 ← GET-BIT-RATIO ( C.child[ 1 ] )
7    else
8        ratio2 ← C.child[ 1 ].type
9    if C.child[ 2 ].type = M
10       ratio3 ← GET-BIT-RATIO ( C.child[ 2 ] )
11    else
12       ratio3 ← C.child[ 2 ].type
13    if C.child[ 3 ].type = M
14       ratio4 ← GET-BIT-RATIO ( C.child[ 3 ] )
15    else
16       ratio4 ← C.child[ 3 ].type
17    return ( ( ratio1 + ratio2 + ratio3 + ratio4 ) / 4 )

```

Below is an example of the scale down operation with a scale factor of half applied on the PM tree of Figure 4. The two stages are shown separately. In the first stage the nodes at lowest levels are removed and the parent values replaced using the GET-BIT-RATIO () function. In the second stage the changes move up the tree modifying the tree structure appropriately.

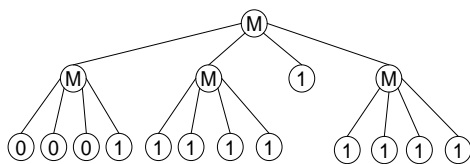


Figure 8. First stage of scale down PM Tree

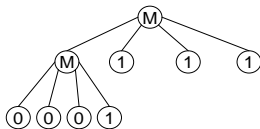


Figure 9. Final scaled down PM tree

The resulting data bit array after the scale down operation is :

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

Figure 10. Data array after scaled down operation

3.2.3. Analysis of the Scaling Algorithms. The time complexity of the algorithms is again measured using the two parameters M , where 2^M is the one of the dimensions of the input binary data, and n , where 2^n is the size of scaling operation. The scale up operation runs in constant time, i.e. $O(1)$ since only the image dimensions need to be updated in the PM tree structure.

However the scale down operation is more complicated. It requires updating all the nodes at level $M - n$ in the PM tree, i.e. $O(4^{M-n})$. Each of these nodes need to call the GET-BIT-RATIO () which basically visits all the nodes from the tree rooted at node passed as the parameter to the GET-BIT-RATIO() function. Hence the GET-BIT-RATIO() has a running time of $O(4^n)$. Hence the total running time of the scale down operation is $O(4^{M-n}) * O(4^n) = O(4^M)$.

3.3. The Ninety Degree Rotation Algorithm

Image rotation is another popular transformation operation along with the translation and scaling operations. It is also the most complicated of the three operations. Usually rotations can be performed over any specified angles. In our paper we have concentrated mainly on rotation operations of ninety degrees either clockwise or anticlockwise.

3.3.1. 90° Clockwise Rotation. As in the previous operations, the ninety degree rotation of an image results in the reordering of the nodes of corresponding PM tree structure of the image. It is evident in the figure below:

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

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

Figure 11. Data array rotated by 90° anticlockwise

We note that the anticlockwise rotation results in recursive reordering of the child nodes in top-down manner through the tree. The child node sequence changes from 0,1,2,3 to 1,3,0,2. A similar reordering occurs when we perform the clockwise rotation.

Table 4. Image rotation algorithm

ROTATE-CLOCKWISE (NODE C)

```

1   if C.child[ 0 ] ≠ null
2       p ← C.child[ 0 ];
3       C.child[ 0 ] ← C.child[ 2 ]
9       C.child[ 2 ] ← C.child[ 3 ]
10      C.child[ 3 ] ← C.child[ 1 ]
11      C.child[ 1 ] ← p
12      ROTATE-CLOCKWISE ( C.child[ 0 ] )
13      ROTATE-CLOCKWISE ( C.child[ 1 ] )
14      ROTATE-CLOCKWISE ( C.child[ 2 ] )
15      ROTATE-CLOCKWISE ( C.child[ 3 ] )

```

ROTATE-ANTICLOCKWISE (NODE C)

```

1   if C.child[ 0 ] ≠ null
2       p ← C.child[ 0 ];
3       C.child[ 0 ] ← C.child[ 1 ]
9       C.child[ 1 ] ← C.child[ 3 ]
10      C.child[ 3 ] ← C.child[ 2 ]
11      C.child[ 2 ] ← p
12      ROTATE-ANTICLOCKWISE ( C.child[ 0 ] )
13      ROTATE-ANTICLOCKWISE ( C.child[ 1 ] )
14      ROTATE-ANTICLOCKWISE ( C.child[ 2 ] )
15      ROTATE-ANTICLOCKWISE ( C.child[ 3 ] )

```

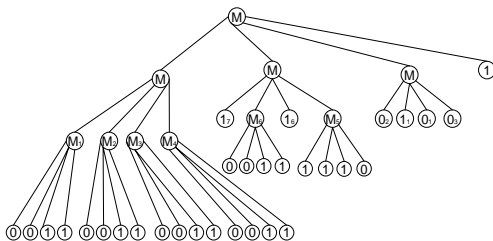


Figure 12. PM-Tree resulting from 90° anticlockwise rotation of the PM-Tree in Figure 4

3.3.2. Analysis of the Rotate Algorithms. Both the rotation algorithms work on all the nodes of the PM tree as they reorder all the child nodes of a single parent node. Hence the time complexity of the rotation algorithms is $O(4^{M-1})$ in the worst case. In more realistic situations where the actual depth and hence number of nodes of the PM tree is smaller, the time complexity is reduced.

4. Experimental results of the Transformation operations

We used a sample image and implemented the above algorithms and obtained the following results:

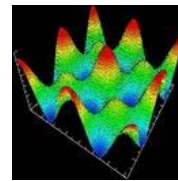


Figure 13. Original picture used



Figure 14. Image scaled down by a factor of 2

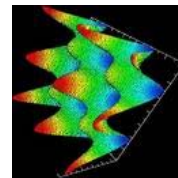


Figure 15. Image rotated by 90° anticlockwise

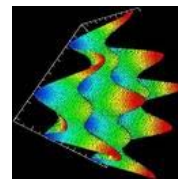


Figure 16. Image rotated by 90° clockwise

5. Conclusion

We are knowledgeable of the fact that the mentioned algorithms can only be used with PM Trees for the shifting of bits. However, variants of Peano

Trees are data mining ready structures that facilitate efficient data mining tasks. These transformation algorithms can be used anywhere PM Trees are being used to recognize patterns. Since PM Trees are both compression and lossless data structures, they can be used in a variety of applications that require lossless storage of their data. Hence, the popularity of proposed algorithm depends on the popularity of the use of the PM Tree data structure.

6. Acknowledgement

Authors of this paper want to acknowledge Dr. William Perrizo, professor, Department of Computer Science, North Dakota State University, Fargo, ND, USA for his continuous support and help to accomplish this work.

7. References

[1] Morgan McGuire, "An image registration technique for recovering rotation, scale and translation parameters". February 19, 1998. Available at

[http://www.cs.brown.edu/people/morganpapers/\(McGuire%20098\)%20Registration.pdf](http://www.cs.brown.edu/people/morganpapers/(McGuire%20098)%20Registration.pdf)

[2] Fazle Rabbi, Mohammad Hossain, et. al. "Lossless Image Compression using P-tree", *Proceedings of ICCIT 2003*, Dhaka, Bangladesh.

[3] Mohammad Hossain, et. al. "Bayesian Classification for Spatial Data Using P-tree", *Proceedings of IEEE INMIC 2004*, December 24-26, 2004 in Lahore, Pakistan.

[4] M.K. Hossain and W. Perrizo, "Automatic fingerprint identification system using p-tree", *Proceedings of ICCIT 2002*, Dhaka, Bangladesh.

[5] William Perrizo, William Jockheck, Amal Perera, "Multimedia Data Mining using P-Trees". Available at http://www-staff.it.uts.edu.au/~simeon/mdm_kdd2002/abstracts/14.html

[6] Quang Ding, Qin Ding and William Perrizo. "Decision tree classification of spatial data streams using peano count trees", *Proceedings of ACM Symposium on Applied Computing (SAC '02)*, Madrid, Spain, March 2002, pp. 413-417.