The P-tree Algebra[1, 2]

Qin Ding, Maleq Khan, Amalendu Roy and William Perrizo
Computer Science Department, North Dakota State University
Fargo, ND 58105-5164, USA
qin.ding@ndsu.nodak.edu

ABSTRACT

The Peano Count Tree (P-tree) is a quadrant-based lossless tree representation of the original spatial data. The idea of P-tree is to recursively divide the entire spatial data, such as Remotely Sensed Imagery 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. In this paper, we will focus on the algebra and properties of P-tree structure and its variations. We have implemented fast algorithms for P-tree generation and P-tree operations. Our performance analysis shows P-tree has small space and time costs compared to the original data. We have also implemented some data mining algorithms using P-trees, such as Association Rule Mining, Decision Tree Classification and K-Clustering.

Keywords

Compression, Quadrant, Peano Ordering, Spatial Data, Tree Structure

1. INTRODUCTION

More and more spatial data have been collected in various ways. An important issue is how to efficiently store spatial data and derive useful information from them. Data mining can help to find interesting patterns or derive useful rules from spatial data. However, existing data mining algorithms do not scale well on large sized spatial data.

In this paper, we try to discuss a data structure, called Peano Count Tree (or P-tree), and its algebra and properties. P-tree structure is a lossless representation of the original spatial data. The P-tree structure can be viewed as a data-mining-ready structure as it facilitates efficient ways for data mining [7].

One feature of spatial data is the neighborhood property. For example, in an image, neighboring pixels may have similar properties. The P-tree structure is based on this feature.

In this paper, we will focus on remotely sensed imagery data, including satellite images, aerial photography, and ground data. A remotely sensed image typically contains several attributes, called bands. For example, TM (Thematic Mapper) scenes contain at least seven bands (Blue, Green, Red, NIR, MIR, TIR and MIR2) while a TIFF image contains three bands (Blue, Green and Red). Each band contains a relative reflectance intensity value in the range 0-to-255 (one byte) for each pixel location. Ground data are collected at the surface of the earth and can also be organized into images. Yield is a typical example of ground data.

An image can be viewed as a relational table in which each pixel is a tuple and each band is an attribute. The primary key can be expressed as *x-y* coordinates or as latitude-longitude pairs. RSI data are collected in different ways and are organized in different formats. BSQ, BIL and BIP are three typical formats. The Band Sequential (BSQ) format is similar to the relational format. In BSQ format, each band is stored as a separate file and each individual band uses the same raster order. TM scenes are in BSQ format. The Band Interleaved by Line (BIL) format stores the data in line-major order, i.e., the first row of all bands, followed by the second row of all bands, and so on. For example, SPOT data, which comes from French satellite sensors, is in BIL format. Band Interleaved by Pixel (BIP) is a pixel-major format. Standard TIFF images are in BIP format.

We propose a new format, called bit Sequential (bSQ), to organize spatial data [7]. A reflectance value in a band is a number in the range 0-255 and is represented as a byte. We split each band into eight separate files, one for each bit position. There are several reasons why we use the bSQ format. First, different bits make different contributions to the value. In some applications, the high-order bits alone provide the necessary information. Second, the bSQ format facilitates the representation of a precision hierarchy (from one bit up to eight bit precision). Third, bSQ format facilitates better compression.

By using bSQ format, neighborhood pixels may have the same bit values in several high order bits. This facilitates high compression for high order bit files and brings us the idea of creating P-trees. P-trees are basically quadrant-wise, Peano-order-run-length-compressed, representations of each bSQ file. There are several operations of P-trees, in which AND is the most important one. The neighborhood feature of image data also makes P-tree operations fast. Operations on a group of pixels having the same bit value can be done together without considering each bit individually. Fast P-tree operations, especially fast AND operation, provide the possibilities for efficient data mining.

In Figure 1, we give a very simple illustrative example with only two bands in a scene having only four pixels (two rows and two columns). Both decimal and binary reflectance values are given. We can see the difference of BSQ, BIL, BIP and bSQ formats.

The rest of the paper is organized as follows. Section 2 summarizes the basic ideas of Peano Count tree and its variation. Section 3 describes the algebra of P-tree. Section 4 discusses the properties of P-tree structure. Section 5 discusses the implementation issues and experimental results. Section 6 gives some related work. Conclusion is given in Section 7.

---

[1] Patents are pending on the bSQ and P-tree technology.

[2] This work is partially supported by GSA Grant ACT# K96130308, NSF Grant OSR-9553368 and DARPA Grant DAAH04-96-1-0329.

SAC 2002, Madrid, Spain

BAND-1	
254	127
(1111 1110)	(0111 1111)
14	193
(0000 1110)	(1100 0001)

BAND-2	
37	240
(0010 0101)	(1111 0000)
200	19
(1100 1000)	(0001 0011)

BSQ format (2 files)

Band 1: 254 127 14 193
Band 2: 37 240 200 19

BIL format (1 file)

254 127 37 240
14 193 200 19

BIP format (1 file)

254 37 127 240
14 200 193 19

bSQ format (16 files, in columns)

B11	B12	B13	B14	B15	B16	B17	B18	B21	B22	B23	B24	B25	B26	B27	B28
1	1	1	1	1	1	1	0	0	0	1	0	0	1	0	1
0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	1	1	1	0	1	1	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	1	0	0	1	1

Figure 1. BSQ, BIP, BIL and bSQ formats for a two-band 2×2 image

2. PEANO COUNT TREES (P-TREES)

2.1 Basic P-trees

We reorganize each bit file of the bSQ format into a tree structure, called a Peano Count Tree (P-tree). The idea is to recursively divide the entire image into quadrants and record the count of 1-bits for each quadrant, thus forming a quadrant count tree [7]. P-trees are somewhat similar in construction to other data structures in the literature (e.g., Quadtrees [3, 4, 5] and HHcodes [6]).

For example, given a 8×8 bSQ file (one-bit-one-band file), its P-tree is as shown in Figure 2.

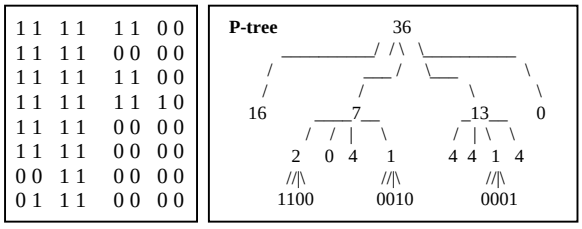

Figure 2. P-tree for a 8×8 bSQ file

In this example, 36 is the number of 1's in the entire image, called root count. This root level is labeled level 0. The numbers 16, 7, 13, and 0 at the next level (level 1) are the 1-bit counts for the four major quadrants in raster order. Since the first and last level-1 quadrants are composed entirely of 1-bits (called pure-1 quadrants) and 0-bits (called pure-0 quadrants) respectively, sub-trees are not needed and these branches terminate. 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 quadrant are pure). If we were to expand all sub-trees, including those for pure quadrants, then the leaf sequence would be the Peano-ordering of the image. The Peano-ordering of the original image is called Peano Sequence. Thus, we use the name Peano Count Tree for the tree structure above.

The fan-out of a P-tree need not be fixed at four. It can be any power of 4 (effectively skipping levels in the tree). Also, the fan-out at any one level need not coincide with the fan-out at another level. The fan-out pattern can be chosen to produce maximum compression for each bSQ file. We use P-Tree-r-i-l to indicate the fan-out pattern, where r is the fan out of the root node, i is the fan out of all internal nodes at level 1 to L-1 (where root has level L, and leaf has level 0), and l is the fan out of all nodes at level 1. We have implemented P-Tree-4-4-4, P-Tree-4-4-16, and P-Tree-4-4-64.

Definition 1: A **basic P-tree** $P_{i,j}$ is a P-tree for the j^{th} bit of the i^{th} band i. The complement of basic P-tree $P_{i,j}$ is denoted as $P_{i,j}'$ (the complement operation is explained below).

For each band (assuming 8-bit data values, though the model applies to data of any number bits), there are eight basic P-trees, one for each bit position. We will call these P-trees the basic P-trees of the spatial dataset. We will use the notation, $P_{b,i}$ to denote the basic P-tree for band, b and bit position, i. There are always 8n basic P-trees for a dataset with n bands.

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).

P-trees can be generated quite quickly and can be viewed as a "data mining ready" and lossless format for storing spatial data.

2.2 P-tree variations

A variation of the P-tree data structure, the Peano Mask Tree (PM-tree, or PMT), is a similar structure in which masks rather than counts are used. In a PM-tree, we use a 3-value logic to represent pure-1, pure-0 and mixed quadrants (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. Since a PM-tree is just an alternative implementation for a Peano Count tree (PC-tree, or PCT), we will use the term "P-tree" to cover both Peano Count tree (PCT) and Peano Mask tree (PMT).

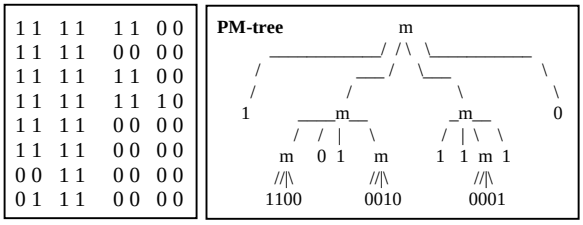

Figure 3. PM-tree

We can use some other variations, such as P1-tree and P0-Tree. In P1-tree, we use 1 to indicate the pure-1 quadrant while use 0 to indicate others. In P0-tree, we use 1 to indicate the pure-0 quadrant while use 0 to indicate others. Both P1-tree and P0-tree are lossless representations of the original data.

The P1-tree and P0-tree of the previous example are given in Figure 4.

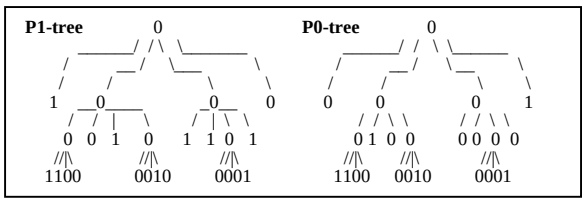

Figure 4. P1-tree and P0-tree

We can also use non-pure-0-tree (NP0-tree) and non-pure-1-tree (NP1-tree). In non-pure-0-tree, we use 1 to indicate non-pure-0 quadrant that covers pure-1 quadrant and mixed quadrant while use 0 to indicate pure-0 quadrant. NP1-tree can be defined in a similar way.

3. P-TREE ALGEBRA

P-tree algebra includes three basic operations: complement, AND and OR. Each basic P-tree has a natural complement. The complement of a basic P-tree can be constructed directly from the P-tree by simply complementing the counts at each level (subtracting from the pure-1 count at that level), as shown in the example below (Figure 4). Note that the complement of a P-tree provides the 0-bit counts for each quadrant. P-tree AND/OR operations are also illustrated in Figure 5.

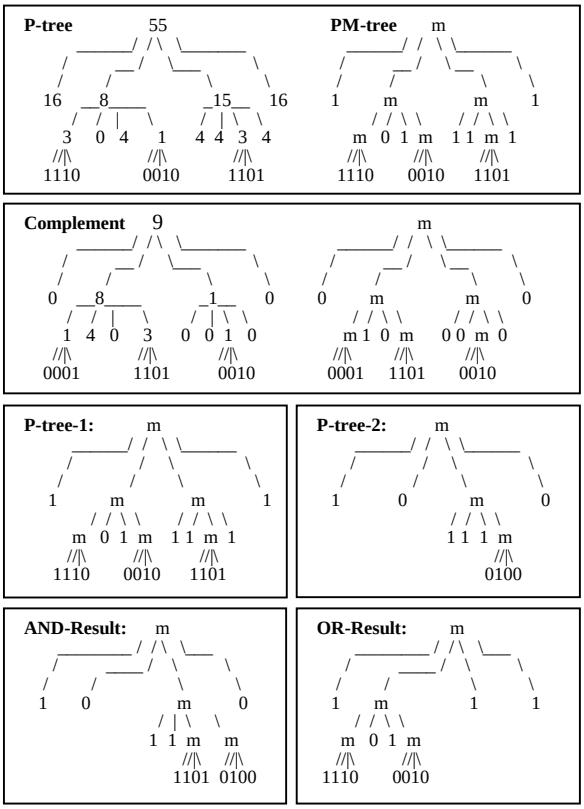

Figure 5. P-tree Algebra (Complement, AND and OR)

Among three operations, AND is the most important operation. OR operation can be implemented in the very similar way as AND. Below we will discuss various options to implement P-tree ANDing.

3.1 Levelwise P-tree ANDing

ANDing is a very important and frequently used operation for P-trees. There are several ways to perform P-tree ANDing. First let's look at a simple way. We can perform ANDing level-by-level starting from the root level. Table 1 gives the rules for performing P-tree ANDing. Operand 1 and Operand 2 are two P-trees (or subtrees) with root X_1 and X_2 respectively. Using PM-trees, X_1 and X_2 could be any value among 1, 0 and m (3-value logic representing pure-1, pure-0 and mixed quadrant). Rules for P-tree ANDing are given in Table 1. For example, to AND a pure-1 P-tree with any P-tree will result in the second operand; to AND a pure-0 P-tree with any P-tree will result in the pure-0 P-tree. Note that it is possible that ANDing two m's results in a pure-0 quadrant if their four subquants result in pure-0 quadrants.

Table 1. P-tree ANDing rules

Operand 1	Operand 2	Result
1	X_2	Subtree with root X_2
0	X_2	0
X_1	1	Subtree with root X_1
X_1	0	0
m	m	0 if four sub-quadrants result in 0; Otherwise m

3.2 P-tree ANDing using Pure-1 paths

There is another way to do P-tree ANDing which is more efficient. The approach is to store only the basic P-trees and then generate the value and tuple P-tree root counts "on-the-fly" as needed. In this algorithm, we will assume P-trees are coded in a compact, depth-first ordering of the paths to each pure-1 quadrant. We use a hierarchical quadrant id (Qid) scheme (Figure 6) to identify quadrants. At each level, we append a sub-quadrant id number (0 means upper left, 1 means upper right, 2 means lower left, 3 means lower right).

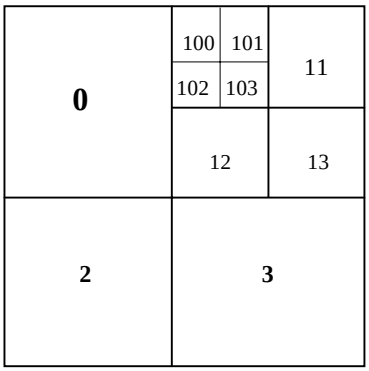

Figure 6. Quadrant id (Qid)

For a spatial data set with 2^n-row and 2^n-column, there is a mapping from raster coordinates (**x**, **y**) to Peano coordinates (called quadrant id or Qid). If **x** and **y** are expressed as n-bit strings, $x_1x_2\ldots x_n$ and $y_1y_2\ldots y_n$, then the mapping is (**x**, **y**)=($x_1x_2\ldots x_n$, $y_1y_2\ldots y_n$) → (x_1y_1 . x_2y_2 **…** . x_ny_n). Thus, in an 8 by 8 image, the pixel at (3,6) = (011,110) has quadrant id 01.11.10 = 1.3.2. For simplicity, we wrote the Qid as 132 instead of 1.3.2 in Figure 6.

An example is given in Figure 7. Each path is represented by the sequence of quadrants in Peano order, beginning just below the root. Since a quadrant will be pure-1 in the result only if it is pure-1 in both/all operands, the AND is done as follows: scan the operands; output matching pure-1 paths.

The AND operation is effectively the pixel-wise AND of bits from bSQ files or their complement files. However, since such files can contain hundreds of millions of bits, shortcut methods are needed. Implementations of these methods have been done which allow the performance of an n-way AND of Tiff-image P-trees (1320 by 1320 pixels) in about 20 milliseconds. We discuss such methods later in the paper. The process of converting data to P-trees is also time consuming unless special methods are used. For example, our methods can convert even a large TM satellite image (approximately 60 million pixels) to its basic P-trees in just a few seconds using a high performance PC computer. This is a one-time process.

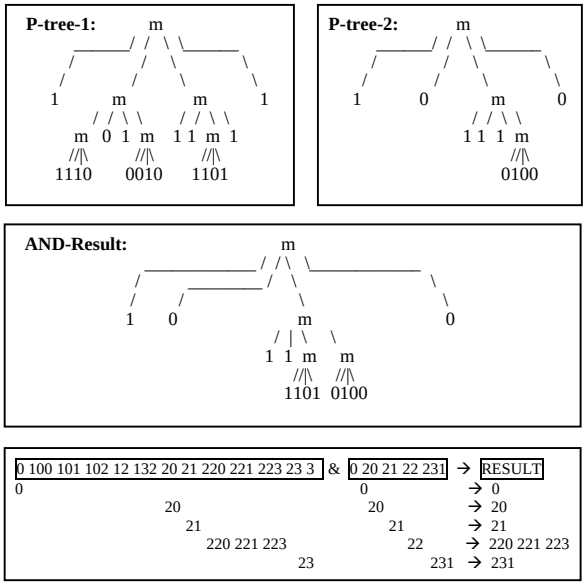

Figure 7. P-tree ANDing using pure-1 path

3.3 Value and Tuple P-trees

By performing the AND operation on the appropriate subset of the basic P-trees and their complements, we can construct P-trees for values with more than one bit.

Definition 2: A **value P-tree** $P_i(v)$, is the P-tree of value v at band i. Value v can be expressed in 1-bit upto 8-bit precision.

Value P-trees can be constructed by ANDing basic P-trees or their complements. For example, value P-tree $P_i(110)$ gives the count of pixels with band-i bit 1 equal to 1, bit 2 equal to 1 and bit 3 equal to 0, i.e., with band-i value in the range of [192, 224). It can be constructed from the basic P-trees as:

$$P_i(110) = P_{i,1} \ \textbf{AND} \ P_{i,2} \ \textbf{AND} \ P_{i,3}'$$

P-trees can also represent data for any value combination from any band, even the entire tuple. In the very same way, we can construct **tuple P-trees**.

Definition 3: A **tuple P-tree** $P(v_1, v_2, \ldots, v_n)$, is the P-tree of value v_i at band i, for all i from 1 to n. We have,

$$P(v_1, v_2, \ldots, v_n) = P_1(v_1) \ \textbf{AND} \ P_2(v_2) \ \textbf{AND} \ \ldots \ \textbf{AND} \ P_n(v_n)$$

If value v_j is not given, it means it could be any value in Band j. For example, P (110, ,101,001, , , ,) stands for a tuple P-tree of value 110 in band 1, 101 in band 3 and 001 in band 4 and any value in any other band.

Definition 4: A **interval P-tree** $P_i(v_1, v_2)$, is the P-tree for value in the interval of $[v_1, v_2]$ of band i. We have,

$$P_i(v_1, v_2) = \textbf{OR} \ P_i(v), \text{ for all v in } [v_1, v_2].$$

Any value P-tree, tuple P-tree can be constructed by performing ANDing on basic P-trees and their complements. Interval P-trees can be constructed by combining AND and OR operations of basic P-trees. All the P-tree operations, including AND, OR, COMPLEMENT and XOR, can be performed on any kinds of P-trees we defined above. The process of ANDing basic P-trees and their complements to produce value P-trees or tuple P-trees can be done at any level of precision -- 1-bit precision up to 8-bit precision.

4. PROPERTIES OF P-TREES

In this section, we will discuss the good properties of P-trees. We will use the following notations:

$p_{x,y}$ is the pixel with coordinate *(x, y)*, $V_{x,y,i}$ is the value for the band *i* of the pixel $p_{x,y}$, $b_{x,y,i,j}$ is the j^{th} bit of $V_{x,y,i}$ (bits are numbered from left to right, $b_{x,y,i,0}$ is the leftmost bit). Indices: *x*: column (*x*-coordinate), *y*: row (*y*-coordinate), *i*: band, *j*: bit.

For any P-trees *P*, P_1 and P_2, P_1 & P_2 denotes P_1 *AND* P_2, $P_1 \mid P_2$ denotes P_1 *OR* P_2, $P_1 \oplus P_2$ denotes P_1 *XOR* P_2, P' denotes COMPLEMENT of *P*.

$P_{i,j}$ is the basic P-tree for bit *j* of band i, $P_i(v)$ is the value P-tree for the value *v* of band i, $P_i(v_1, v_2)$ is the interval P-tree for the interval $[v_1, v_2]$ of band *I*, *rc(P)* is the root count of P-tree *P*. P^0 is *pure*-0 *tree*, P^1 is *pure*-1 *tree*. *N* is the number of pixels in the image or space under consideration.

Lemma 1: *For any two P-trees P_1 and P_2, $rc(P_1 \mid P_2) = 0 \Rightarrow rc(P_1) = 0$ and $rc(P_2) = 0$.* More strictly, *$rc(P_1 \mid P_2) = 0$, if and only if $rc(P_1) = 0$ and $rc(P_2) = 0$.*

Proof: (Proof by contradiction) Let, *$rc(P_1) \neq 0$.* Then, for some pixels there are 1s in P_1 and for those pixels there must be 1s in $P_1 \mid P_2$ i.e. *$rc(P_1 \mid P_2) \neq 0$*, But we assumed *$rc(P_1 \mid P_2) = 0$*. Therefore *$rc(P_1) = 0$*. Similarly we can prove that *$rc(P_2) = 0$*.
The proof for the inverse, *$rc(P_1) = 0$ and $rc(P_2) = 0 \Rightarrow rc(P_1 \mid P_2) = 0$* is trivial. This immediately follows the definitions.

Lemma 2: a) *$rc(P_1) = 0$ or $rc(P_2) = 0 \Rightarrow rc(P_1 \ \& \ P_2) = 0$*
b) *$rc(P_1) = 0$ and $rc(P_2) = 0 \Rightarrow rc(P_1 \ \& \ P_2) = 0$.*
c) *$rc(P^0) = 0$*
d) *$rc(P^1) = N$*
e) *$P \ \& \ P^0 = P^0$*
f) *$P \ \& \ P^1 = P$*
g) *$P \mid P^0 = P$*
h) *$P \mid P^1 = P^1$*
i) *$P \ \& \ P' = P^0$*
j) *$P \mid P' = P^1$*

Proofs are immediate.

Lemma 3: *$v_1 \neq v_2 \Rightarrow rc\{P_i(v_1) \ \& \ P_i(v_2)\} = 0$, for any band i.*

Proof: $P_i(v)$ represents all the pixels having value v for the band i. If $v_1 \neq v_2$, no pixel can have the values of both v_1 and v_2 for the same band. Therefore, if there is a 1 in $P_i(v_1)$ for any pixel, there must be 0 in $P_i(v_2)$ for that pixel and vice versa. Hence $rc\{P_i(v_1) \& P_i(v_2)\} = 0$.

Lemma 4: $rc(P_1 \mid P_2) = rc(P_1) + rc(P_2) - rc(P_1 \& P_2)$.

Proof: Let the number of pixels for which there are 1s in P_1 and 0s in P_2 is n_1, the number of pixels for which there are 0s in P_1 and 1s in P_2 is n_2 and the number of pixels for which there are 1s in both P_1 and P_2 is n_3.

Now, $rc(P_1) = n_1 + n_3$, $rc(P_2) = n_2 + n_3$, $rc(P_1 \& P_2) = n_3$
and $rc(P_1 \mid P_2) = n_1 + n_2 + n_3 = (n_1 + n_3) + (n_2 + n_3) - n_3$
$= rc(P_1) + rc(P_2) - rc(P_1 \& P_2)$

Theorem: $rc\{P_i(v_1) \mid P_i(v_2)\} = rc\{P_i(v_1)\} + rc\{P_i(v_2)\}$, *where* $v_1 \neq v_2$.

Proof: $rc\{P_i(v_1) \mid P_i(v_2)\} = rc\{P_i(v_1)\} + rc\{P_i(v_2)\} - rc\{P_i(v_1) \& P_i(v_2)\}$ (Lemma 4)
If $v_1 \neq v_2$, $rc\{P_i(v_1) \& P_i(v_2)\} = 0$. (Lemma 3)
Therefore, $rc\{P_i(v_1) \mid P_i(v_2)\} = rc\{P_i(v_1)\} + rc\{P_i(v_2)\}$.

5. IMPLEMENTATION ISSUES AND EXPERIMENTAL RESULTS

5.1 P-tree Header

To make a generalized P-tree structure, the following header for a P-tree file is proposed in table 2.

Table 2. P-tree header

1 byte	2 bytes	1 byte	4 bytes	2 bytes	
Format Code	Fan-out	# of levels	Root count	Length of the body	Body of the P-tree

Format code: Format code identifies the format of the P-tree, whether it is a *PCT* or *PMT* or in any other format.

Fan-out: This field contains the fan-out information of the P-tree. Fan-out information is required to traverse the P-tree in performing various P-tree operations.

# of levels: Number of levels in the P-tree. When we encounter a *pure1* or *pure0* node, we cannot tell whether it is an interior node or a leaf unless we know the level of that node and the total number of levels of the tree. This is required to know the number of 1s represented by a *pure1* node.

Root count: *Root count* i.e. the number of 1s in the P-tree. Though we can calculate the *root count* of a P-tree on the fly from the P-tree data, these only 4 bytes of space can save computation time when we only need the root count of a P-tree. The root count of a P-tree can be computed at time of construction of P-tree with a very little extra cost.

Length of the body: Length of the body is the size of the P-tree file in bytes excluding the header. Sometimes we may want to keep the whole P-tree into RAM to increase the efficiency of computation. Since the size of the P-tree varies, we need to allocate memory dynamically, which requires knowing the size of the required memory size before reading the data from disk.

5.2 Performance Analysis

We only store the basic P-trees for each dataset. All other P-trees (value P-trees and tuple P-trees) are created on the fly as needed. This results in a considerable saving of space. Figure 8, 9 and 10 give the storage needs for various formats of data (TIFF, SPOT and TM scene) using various formats of P-trees (PCT or PMT) with different fan-out patterns.

The AND operation of basic P-trees is fast. Figure 11 and 12 show the time required to perform P-tree ANDing. The ANDing time varies from 6.72 ms to 52.12 ms for two TM scenes with size 2048×2048.

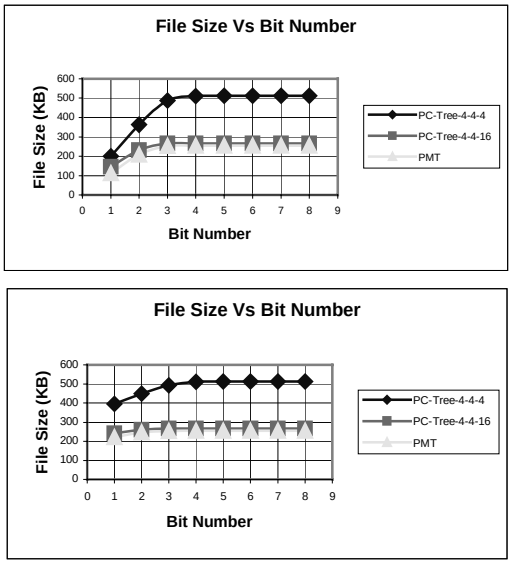

Figure 8. Comparison of file size for different bits of Band 1 & 2 of a TIFF image

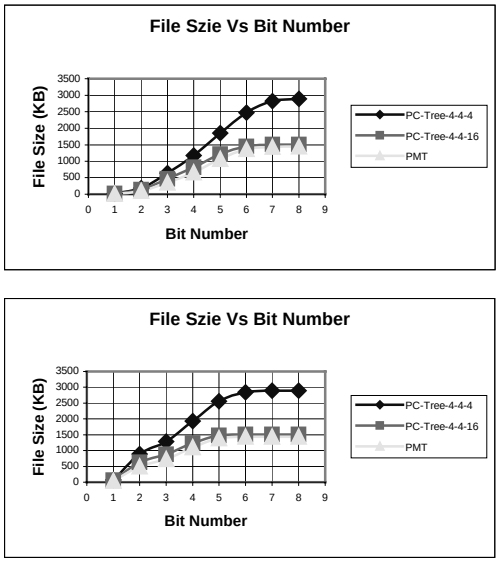

Figure 9. Comparison of file size for different bits of Band 3 & 4 of a SOPT image

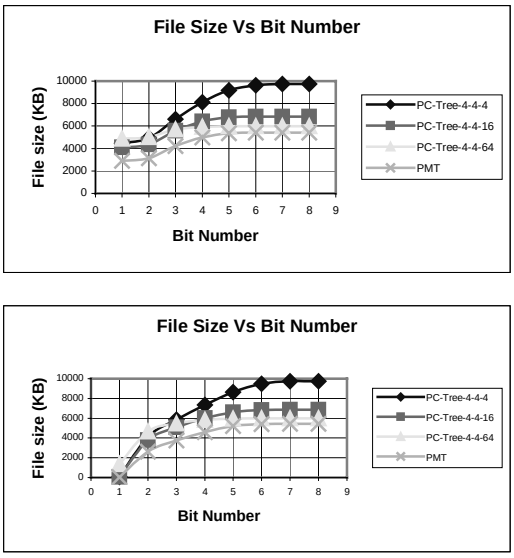

Figure 10. Comparison of file size for different bits of Band 5 & 6 of a TM image

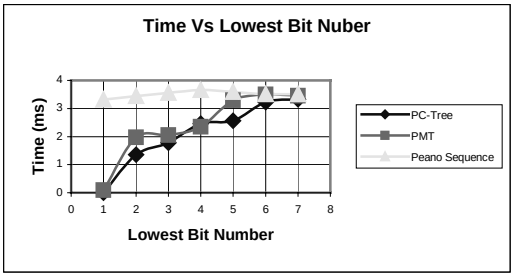

Figure 11. Comparison of time required to perform ANDing operation

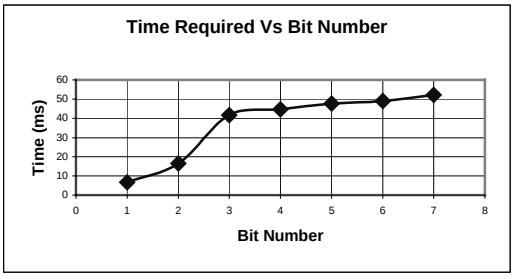

Figure 12. Average time required to perform ANDing operation

6. RELATED WORK

Concepts related to the P-tree data structure, include Quadtrees [1, 2, 3, 4, 5] and its variants (such as point quadtrees [3] and region quadtrees [4]), and HH-codes [6].

Quadtrees decompose the universe by means of iso-oriented hyperplanes. These partitions do not have to be of equal size, although that is often the case. The decomposition into subspaces is usually continued until the number of objects in each partition is below a given threshold. Quadtrees have many variants, such as point quadtrees and region quadtrees.

HH-codes, or Helical Hyperspatial Codes, are binary representations of the Riemannian diagonal. The binary division of the diagonal forms the node point from which eight sub-cubes are formed. Each sub-cube has its own diagonal, generating new sub-cubes. These cubes are formed by interlacing one-dimensional values encoded as HH bit codes. When sorted, they cluster in groups along the diagonal. The clusters are order in a helical pattern, thus the name "Helical Hyperspatial".

The similarities among P-tree, quadtree and HHCode are that they are quadrant based. The difference is that P-trees focus on the count. P-trees are not index, rather they are representations of the datasets themselves. P-trees are particularly useful for data mining because they contain the aggregate information needed for data mining.

7. CONCLUSION

In this paper, we describe the properties and algebra of a lossless data structure, Peano Count Tree (P-tree). P-tree structure has nice features so that it can be used for efficient data mining, such as association rule mining, classification and clustering. The details of how to use P-trees in data mining is beyond the scope of this paper. The basic idea is that in the processing of data mining, a lot of counts are needed. With the information in P-trees, these counts can be collected in a fast way by ANDing appropriate P-trees instead of scanning the entire database. The idea of P-tree initially came from the spatial data, however, it can be extended to represent other kinds of data, such as DNA Microarray data and VLSI data.

8. REFERENCES

[1] Volker Gaede and Oliver Gunther, “Multidimensional Access Methods”, Computing Surveys, 30(2), 1998.

[2] H. Samet, “The quadtree and related hierarchical data structure”. ACM Computing Survey, 16, 2, 1984.

[3] H. Samet, “Applications of Sptial Data Structures”, Addison-Wesley, Reading, Mass., 1990.

[4] H. Samet, “The Design and Analysis of Spatial Data Structures”, Addison-Wesley, Reading, Mass., 1990.

[5] R. A. Finkel and J. L. Bentley, “Quad trees: A data structure for retrieval of composite keys”, Acta Informatica, 4, 1, 1974.

[6] HH-codes. Available at http://www.statkart.no/nlhdb/iveher/hhtext.html

[7] William Perrizo, Qin Ding, Qiang Ding and Amalendu Roy, “Deriving High Confidence Rules from Spatial Data using Peano Count Trees”, Springer-Verlag, LNCS 2118, July 2001.