EPiC Series in Computing
Volume 58, 2019, Pages 45–54
Proceedings of 34th International Conference on Computers and Their Applications

A New Method of Calculating Squared Euclidean Distance (SED) Using pTree Technology and Its Performance Analysis

Mohammad K. Hossain, Sameer Abufardeh
Department of Math, Science and Technology
University of Minnesota Crookston, Crookston, MN 56716, USA
hossain@crk.umn.edu, sabufard@crk.umn.edu

Abstract

One of the advantages of Euclidean distance is that it measures the regular distance between two points in space. For this reason, it is widely used in the applications where the distance between data points are needed to be calculated to measure similarities. However, this method is costly as there involve expensive square and square root operations. One useful observation is that in many data mining applications absolute distance measures are not necessary as long as the distances are used to compare the closeness between various data points. For example, in classification and clustering, we often measure the distances of multiple data points to compare their distances from known classes or from centroids to assign those points in a class or in a cluster. In this regards, an alternative approach known as Squared Euclidean Distance (SED) can be used to avoid the computation of square root to get the squared distance between the data points. SED has been used in classification, clustering, image processing, and other areas to save the computational expenses. In this paper, we show how SED can be calculated for the vertical data represented in pTrees. We also analyze its performance and compared it with traditional horizontal data representation.

1 Introduction

The general distance between any two points in an *n*-dimensional space is measured by weighted Minkowski distance. Considering two points, X and Y, in n-dimensional space as a vector $<x_1, x_2, x_3, \ldots, x_n>$ and $<y_1, y_2, y_3, \ldots, y_n>$, the weighted Minkowski distance between the points is,

$$d_p(X,Y) = \left\{ \sum_{i=1}^{n} w_i \left| x_i - y_i \right|^p \right\}^{\frac{1}{p}} \tag{1}$$

where p is a positive integer, x_i and y_i are the i^{th} components of X and Y, respectively, w_i (≥ 0) is the weight associated with the i^{th} dimension or i^{th} feature.

When the weights w_i's are equal to 1 and p is 2, the Minkowski distance is known as the Euclidian distance or L_2 distance [1], which is:

$$d_2(X,Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{2}$$

From (2) Squared Euclidean Distance (or SED) is calculated by taking the square of the right-hand side and is defined as:

$$SED(X,Y) = \sum_{i=1}^{n}(x_i - y_i)^2 \tag{3}$$

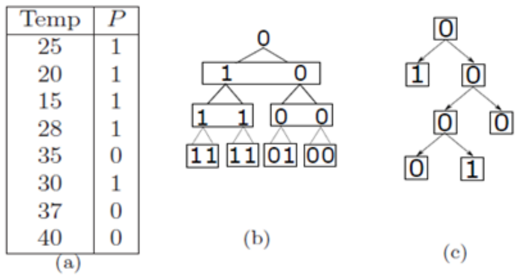
In this paper, we assume that the data are represented in a vertical form using a special data structure called predicate tree or pTree, which is a lossless, compressed, and data-mining-ready data structure [6]. In [2] Manhattan distance was calculated using pTrees. In [3, 4], the summation of SED's of a set of values is calculated rather than the individual SED's. However, in many classification, clustering and other data mining algorithms we need to compute SED values themselves, not just their sum [7, 8]. In [5], authors attempted to calculate SED's but the detail performance analysis was missing in that paper. In addition, the algorithm in [5] uses complex stack operations in to calculate SED, which will make the calculation very slow comparing with our proposed method. In this proposed method, we are able to calculate SED values without any horizontal scanning of the data points, which makes the method suitable for many data mining algorithms.

In section 2, we reviewed the pTree technology to give the readers necessary background knowledge regarding pTree technology. In section 3, we described the algorithms that are used to calculate SED. Then in section 4, we discussed the experiments we carried out and analyzed the results.

2 Review of pTree Technology

The predicate tree or pTree is a vertical data technology, which records the truth-value of a given predicate on a given data set. For each value of the data set, it stores 1 if the given predicate is true and stores 0 if the predicate is false. Thus with these 1's and 0's, we get a bit vector which forms the leaf of the tree. Next, we group the bits of the leaf with a fixed number of bits and apply some other predicate to each group to represent them by 1 or 0 (based on the truth value of the predicate) to form the parent node of that group. In a similar fashion, we group the bits the parent nodes to form upper-level parent nodes. This process continues until we form the root node containing a single bit. Next, we examine the tree to see if all the bits in a group is all 1's (called pure-1 node) or all 0's (called pure-0 node). In that case, we remove all its child nodes.

To illustrate the construction of a pTree, consider the example in Figure 1 (a) where the temperature of a city is recorded for 8 days and a predicate P chosen as "the weather is freezing" (i.e. below 32). Therefore, for the first four data value P is true, for the next one it is false and so on. Thus, we get a bit vector "11110100" which becomes the leaf of the pTree as shown in Figure 1 (b). As we see in Figure 1 (b), the bits of the leaf node are grouped with two bits and the parent nodes are formed with a predicate "the group contains all 1". The process goes on until we form the root of the tree. Finally, we discard the children nodes that are pure-1 or pure-0 nodes. Figure 1(c) shows the final pTree.



Temp	P
25	1
20	1
15	1
28	1
35	0
30	1
37	0
40	0

Figure 1: Construction of pTree: (a) Predicate for temperature (b) Conversion to tree (c) Final pTree

2.1 Boolean Algebra for pTrees

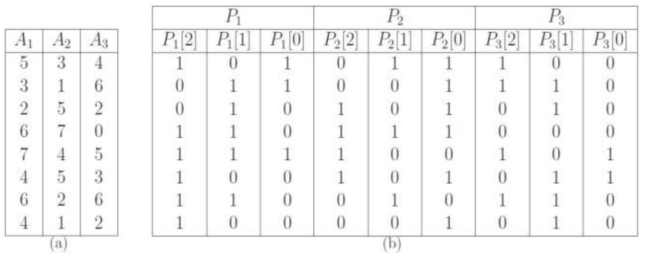
Boolean algebra for a set of two elements B = {0, 1} is generally defined with 1 being the TRUE state and 0 being the FALSE state. The three basic operations i.e. AND, OR and NOT are explained with two sets X and Y $\in$ {0, 1}. [9]

2.2 Important Definitions

Definition 1 (Level-0 pTree): In the process of creating a pTree, the bit slice that is created by representing the truth-value of a predicate by 1 or 0 is known as level-0 pTree. The number of bits (0 or 1) in a level-0 pTree is known as the length of pTree. A level-0 pTree is sometimes mentioned only by pTree. The leaf of pTree in Figure 1 (b) is an example of a level-0 pTree.

Definition 2 (pTree set): A pTree set P of size N is a collection of N pTrees where each pTree in the collection is accessed by $P[i]$ where $0 \leq i \leq (N-1)$. The number of pTrees in a pTree set is known as the size of the pTree set.

Figure 2 (b) shows three pTree sets *P1*, *P2* and *P3* constructed from the data set of Figure 2 (a). Size of each pTree set is 3.



A_1	A_2	A_3
5	3	4
3	1	6
2	5	2
6	7	0
7	4	5
4	5	3
6	2	6
4	1	2

(a)

P_1			P_2			P_3		
$P_1[2]$	$P_1[1]$	$P_1[0]$	$P_2[2]$	$P_2[1]$	$P_2[0]$	$P_3[2]$	$P_3[1]$	$P_3[0]$
1	0	1	0	1	1	1	0	0
0	1	1	0	0	1	1	1	0
0	1	0	1	0	1	0	1	0
1	1	0	1	1	1	0	0	0
1	1	1	1	0	0	1	0	1
1	0	0	1	0	1	0	1	1
1	1	0	0	1	0	1	1	0
1	0	0	0	0	1	0	1	0

(b)

Figure 2: pTree representation of Data Set: (a) Data Set of 3 attributes (b) Corresponding pTree Sets

2.3 pTree Representation of Dataset

The task of generating pTree's typically starts by converting a relational table of horizontal records to a set of vertical bit vectors by decomposing each attribute in the table into separate bit slices. If an attribute has numeric values, we convert the data into binary and then consider the predicates to be "2^0 position is 1", "2^1 position is 1", "2^2 position is 1" and so on. Then each bit position of the binary value generates a bit slice. But if an attribute has categorical value then first we need to create a bitmap for the attribute and then generate bit vectors for each category. Such vertical partitioning guarantees that the information is not lost.

Consider a dataset ***D*** with *n* attributes *as* $\boldsymbol{D} = (A_1, A_2, \ldots, A_n)$. In order to represent it by pTree we will require a pTree set of size *n* as $\{P_1, P_2 \ldots P_n\}$ such that attribute A_i will be represented by pTree set P_i. Suppose each value of A_i is represented by an *N*-bit binary number $a_{i,N-1}, a_{i,N-2} \ldots, a_{i,j}, \ldots, a_{i,0}$. Then pTree $P_{i,}[j]$ will represent the bit slice of $a_{i,j}$. $P_i[0]$ pTree which represent the Least significant bit slice of Ai is called the lest significant pTree or **LSP** of pTree set P_i. Similarly, $P_i[N-1]$ is known as the most significant pTree or **MSP**. Figure 2 shows the representation of data set into pTree where we see that the dataset had three attributes. Hence, we need three pTree sets to represent the dataset. Observing the values of the attributes, we see that we need 3-bit numbers to convert them to binary. Therefore, the size of each pTree set is 3.

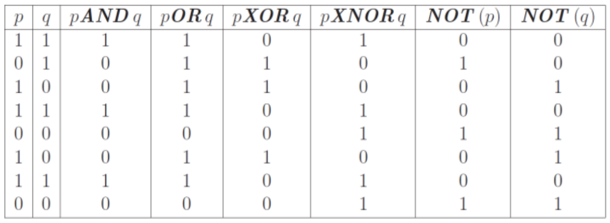
2.4 Operations on pTrees

Suppose *p* and *q* are two level-0 pTrees of length *L*. Assume a binary operator $\mathbf{OP_b}$ that we apply on these two pTrees. Then p $\mathbf{OP_b}$ q is calculated as:

$p[i]$ $\mathbf{OP_b}$ $q[i]$ $\forall i | i \ni (0 : L-1)$ where $p[i]$and $q[i]$ are the *i*th bit of *p* and *q* pTrees.

Similarly, if $\mathbf{OP_u}$ is a unary operator then $\mathbf{OP_u}(p)$ is calculated as $\mathbf{OP_u}(p[i])$.

Let *p* = 10110110 and *q* = 11010010 are two level-0 pTrees. Figure 3 shows the results of different operations on these pTrees.



p	q	$p\boldsymbol{AND}q$	$p\boldsymbol{OR}q$	$p\boldsymbol{XOR}q$	$p\boldsymbol{XNOR}q$	$\boldsymbol{NOT}(p)$	$\boldsymbol{NOT}(q)$
1	1	1	1	0	1	0	0
0	1	0	1	1	0	1	0
1	0	0	1	1	0	0	1
1	1	1	1	0	1	0	0
0	0	0	0	0	1	1	1
1	0	0	1	1	0	0	1
1	1	1	1	0	1	0	0
0	0	0	0	0	1	1	1

Figure 3: Truth table showing different pTree

3 Necessary Algorithms for SED

In various steps of SED calculation, we will require to perform many operations on pTree sets. For instance, we will need to subtract a constant value from a pTree set. Then we will need to calculate the absolute value of that resultant pTree set. Then we will compute the square of that pTree set by multiplying it with itself. Finally, we will add these pTree sets to get the SED. In the following subsections, we describe these algorithms.

3.1 Finding Absolute pTree set

When we subtract an integer from another, the result can be positive or negative depending on the value of the integers. The negative results are shown in 2's complement form. In order to get the value of a negative result, we need to get the absolute value of that negative result. The algorithm in Figure 4 first examines the rightmost pTree of the input pTree set that is considered as sign pTree. When the value of that pTree is 0 the number represented is positive and otherwise negative. When it is a positive value, nothing is done. However, when it is 1 it indicates that the number is negative and then the number is complemented to get the absolute value.

3.2 Adding Two pTree sets

When adding two pTree sets we add them and store the result in a third pTree set. Figure 5 shows the addition of two pTree sets. In the algorithm, we consider two pTree sets of unequal sizes *N* and *M* where *N* > *M*. The summation is stored in the pTree set *S* that has *N + 1* pTrees. Line 2 shows the looping through LSP to MSP of pTree set of size *N*. While adding two pTree sets we deal with two pTrees to calculate sum and carry (line 4 and 5) up to *M*th pTree. For the rest of the pTrees we add only the carry *c* (in line 7 and 8). Finally, the carry c is assigned to *S[N]* in line 11.

```
Algorithm    Compute the absolute value of a pTree set
Input:
pTree Set $A$ of $N+1$ pTrees
Output:
pTree Set $S$ of $N$ pTrees
Variables:
pTree $c, t$
integer $i$

AbspTreeSet($A, S$):
1.    $c \leftarrow a[N]$
2.    foreach $i$ in $(0 : N-1)$ do
3.       $t \leftarrow a[i]\,\boldsymbol{XOR}\,a[N]$
4.       $s[i] \leftarrow t\,\boldsymbol{XOR}\,c$
5.       $c \leftarrow t\,\boldsymbol{AND}\,c$
6.    endfor
```

Figure 4: Algorithm to compute the absolute value of a pTree set operations.

```
Algorithm    Computing the addition of two pTree sets
Input:
pTree Set $A$ of $N$ pTrees
pTree Set $B$ of $M$ pTrees
   where $N > M$
Output:
pTree Set $S$ of $N+1$ pTrees
Variables:
pTree $c$
integer $i$

AddpTreeSet($A, B, S$):
1.    $c \leftarrow 0$
2.    foreach $i$ in $(0 : N-1)$ do
3.       if $i > M$
4.          $S[i] \leftarrow A[i]\,\boldsymbol{XOR}\,B[i]\,\boldsymbol{XOR}\,c$
5.          $c \leftarrow (A[i]\,\boldsymbol{AND}\,B[i])\,\boldsymbol{OR}\,(A[i]\,\boldsymbol{XOR}\,B[i])\,\boldsymbol{AND}\,c$
6.       else
7.          $S[i] \leftarrow A[i]\,\boldsymbol{XOR}\,c$
8.          $c \leftarrow A[i]\,\boldsymbol{AND}\,c$
9.       endif
10.   endfor
11.   $S[N] \leftarrow c$
```

Figure 5: Algorithm to add two pTree sets operations.

3.3 Subtracting a Constant from a pTree set

Subtraction is very much similar to addition with few differences in the equation used. Figure 6 shows the subtraction of a constant from a pTree set. The subtraction result is stored in the pTree set *S* which has *N + 1* pTrees. The result will be a signed integer in 2's complement form. That is the *S[N]* pTree is a sign pTree (similar to a sign bit in a signed integer). A 0 value in this pTree indicates the row in that S is a positive integer and a 1 indicates the same as a negative integer.

```
Algorithm    Subtraction of a constant from a pTree set
Input:
pTree Set $A$ of $N$ pTrees
integer $V$ of $M$ bits, where $N > M$
Output:
pTree Set $S$ of $N{+}1$ pTrees
Variables:
pTree $c$
integer $i$

SubpTreeSet$(A, V, S)$:
1.  $c \leftarrow 0$
2.  foreach $i$ in $(0 : N-1)$ do
3.    if $i > M$ And $V[i] = 1$
4.      $S[i] \leftarrow A[i]\,\boldsymbol{XOR}\,c$
5.      $c \leftarrow \boldsymbol{NOT}\,(A[i])\,\boldsymbol{OR}\,c$
6.    else
7.      $S[i] \leftarrow \boldsymbol{NOT}\,(A[i])\,\boldsymbol{XOR}\,c$
8.      $c \leftarrow \boldsymbol{NOT}\,(A[i])\,\boldsymbol{AND}\,c$
9.    endif
10. endfor
11. $S[N] \leftarrow c$
```

Figure 6: Algorithm to subtract a constant from a pTree set

3.4 Multiplying Two pTree sets

The multiplication algorithm in Figure 7 multiplies pTree set *A* of size *N* by pTree set *B* of size *M* and the result is stored in pTree set *S* of size *M + N*. In line 1, the algorithm loops through all the pTrees of *B* from LSP to MSP. In each iteration, it loops through the pTrees of *A* (as in line 3), multiplies the pTrees of *A* by a single pTree of *B* namely *B[j]* (line 4). Then the algorithm adds this product with *S* (line 5, 6 and 7). In the next iteration, the algorithm multiplies each pTree of *A* by the next pTree of *B*. Then it adds this product with *S* after shifting the result one place to the left. The shifting is done by the assignment of the variable *i* in the loop in line 3.

3.5 Computing SED

Suppose a data set *S* has *n* attributes. An attribute A_i is represented by a pTree set P_i containing N pTrees. That is $S = \{P_1, P_2, \ldots, P_n\}$. Assume we need to find the SED of all the points of the data set *S* from a fixed point $C = (c_1, c_2, \ldots, c_n)$. Suppose the distance will be calculated in pTree set *D*. This will be done by the following steps for all the values of *i* from 1 to *n*.

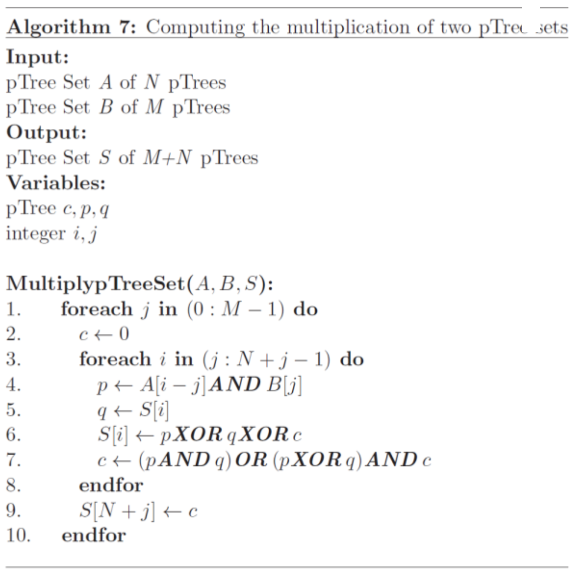
Step 1: Subtract c_i from pTree set P_i using the algorithm in Figure 6. Assume the resultant pTree set is R_i.

Step 2: Get the absolute value pTree of R_i using the algorithm in Figure 4. Assume the absolute value pTree is T_i

Step 3: Get the squared value of T_i by multiplying it with itself using the algorithm in Figure 7. Assume the squared value pTree is S_i

Step 4: Add S_i with pTree set D using the algorithm in Figure 5.

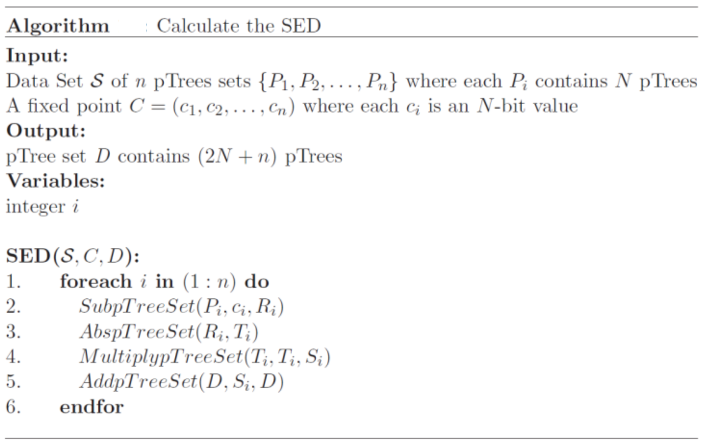
The algorithm in Figure 8 shows these steps.



```
Algorithm 7: Computing the multiplication of two pTre sets
Input:
pTree Set $A$ of $N$ pTrees
pTree Set $B$ of $M$ pTrees
Output:
pTree Set $S$ of $M+N$ pTrees
Variables:
pTree $c, p, q$
integer $i, j$

MultiplypTreeSet($A, B, S$):
1.   foreach $j$ in $(0 : M-1)$ do
2.     $c \leftarrow 0$
3.     foreach $i$ in $(j : N+j-1)$ do
4.       $p \leftarrow A[i-j]\ \boldsymbol{AND}\ B[j]$
5.       $q \leftarrow S[i]$
6.       $S[i] \leftarrow p\ \boldsymbol{XOR}\ q\ \boldsymbol{XOR}\ c$
7.       $c \leftarrow (p\ \boldsymbol{AND}\ q)\ \boldsymbol{OR}\ (p\ \boldsymbol{XOR}\ q)\ \boldsymbol{AND}\ c$
8.     endfor
9.     $S[N+j] \leftarrow c$
10.  endfor
```

Figure 7: Algorithm to multiply two pTree sets



```
Algorithm : Calculate the SED
Input:
Data Set $\mathcal{S}$ of $n$ pTrees sets $\{P_1, P_2, \ldots, P_n\}$ where each $P_i$ contains $N$ pTrees
A fixed point $C = (c_1, c_2, \ldots, c_n)$ where each $c_i$ is an $N$-bit value
Output:
pTree set $D$ contains $(2N+n)$ pTrees
Variables:
integer $i$

SED($\mathcal{S}, C, D$):
1.  foreach $i$ in $(1 : n)$ do
2.    $SubpTreeSet(P_i, c_i, R_i)$
3.    $AbspTreeSet(R_i, T_i)$
4.    $MultiplypTreeSet(T_i, T_i, S_i)$
5.    $AddpTreeSet(D, S_i, D)$
6.  endfor
```

Figure 8: Algorithm to calculate SED

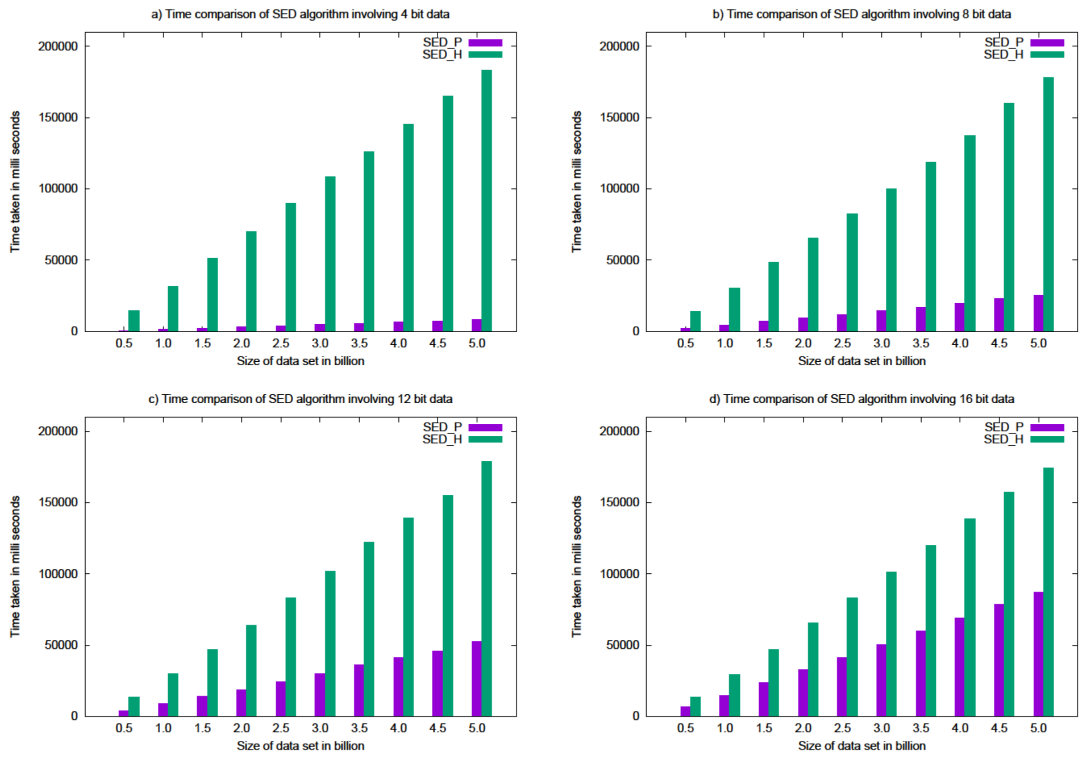


Figure 9: Result of the experiment for data width from 4 to 16 bit.

4 Experimental Result and Analysis

In order to prove the effectiveness of our method of calculating squared Euclidean distance (SED) over traditional horizontal method, we ran a series of experiment using a comprehensive experimental framework based on formal design methodology [10, 11].

In our experiment, we calculated the SED of a set of points from a random point. We assumed the points are in three-dimensional space. Therefore, we have three attributes in the data set, which are represented by three pTree sets. Using the algorithm as shown in Figure 8 (SED_P), we calculated SED.

Then we calculated the SED of the same dataset using the traditional horizontal algorithm (SED_H). We ran these algorithms for each data set cardinality of 0.5 billion to 5.0 billion with 0.5 billion increments.

We also changed the bit width of the data starting from 4 bits up to 32bits with 4 bits increment. Thus, we got 160 design points, which we ran 10 times each and got their average.

We represent the results of our experiment in histograms. For each of the bit width, we have one histogram. Each histogram shows the size of the data set in x-axis and the response time in millisecond in y-axis. We also showed the speed gain (or speed loss) of pTree processing over horizontal processing

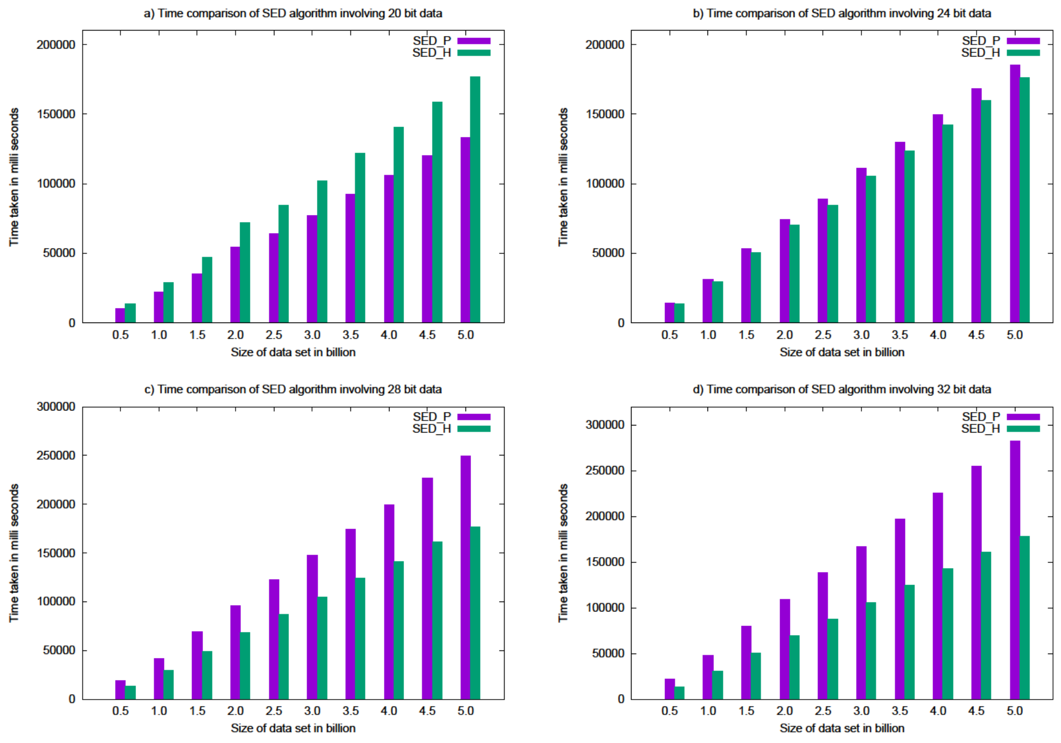


Figure 10: Result of the experiment for data width from 20 to 32 bit.

for each bit width. To measure the speed gain we use the following formula

$$Speed\ gain = (1 - \frac{T_P}{T_H})$$

Where T_P is the time taken for pTree processing and T_H is the time taken for horizontal processing.

The Figure 9 and 10 shows the result of experiment. Table 1 shows the speed gain of SED calculation algorithm. As we can see from the table, we get a 96% of speed gain over horizontal processing for bit width of 4. We get 24% of speed gain for bit width of 20. However, after that we get negative speed gain of 5% for bit width of 24. Therefore, we can say up to bit width of 20 pTree processing is faster than horizontal processing.

Bit Width	Speed Gain (%)
4	96
8	86
12	71
16	50
20	24
24	-5

Table 1: Speed gain of pTree based SED calculation algorithm

5 Conclusion

In this paper, we have demonstrated an effective method to calculate SED using pTrees. Our experimental results exhibit that up to 20-bit numbers, the proposed method has significant speed gain over traditional horizontal method. For the 4-bit numbers, we got most speed gains of 96%, which decreased to 24% for 20-bit numbers. In the case of 24-bit numbers, the method becomes a little bit slower than the traditional method (-5%). Hence, we may use this method up to 20-bit numbers. Using 20-bit numbers, we can represent integer number as large as 1,048,575, which would be sufficient for most data mining algorithms. Our experiment shows similar speed gain for different cardinality of the datasets ranging from 0.5 billion to 5 billion. The speed gain makes our method scalable to large datasets, which is a critical issue for big-data processing. Therefore, we can conclude that the proposed method will be applicable in the "Big Data" processing algorithms where distance or similarity between data points are computed using Squared Euclidean Distance (SED).

References

[1] Maleq Khan, Master thesis on Fast Distance Metric Based Data Mining Techniques Using P-trees, Department of Computer Science, NDSU, 2001.

[2] Mohammad K Hossain, Arjun Roy, Arijit Chatterjee, William Perrizo, "Algorithms to Calculate the Manhattan (L1) Distance for Vertical Data Represented in pTrees", published in the Conference Proceedings at the ISCA 27th International Conference on Computers and Their Applications (CATA-2012), held on March 12-14, 2012, at Las Vegas, Nevada, USA.

[3] A Denton, Q. Ding, W. Pernzo and Q. Ding, Efficient hierarchical clustering of large data sets using P-trees, In Proceedings of the 15th International Conference on Computer Applications in Industry and Engineering (CAINE'02), San Diego, CA, November 2002, pp. 138-141.

[4] T. Abidin, A. Perera, M. Serazi, W. Perrizo, Vertical Set Square Distance: A Fast and Scalable Technique to Compute Total Variation in Large Datasets, CATA-2005 New Orleans, 2005.

[5] Mohammad K Hossain, Arijit Chatterjee, Arjun Roy, William Perrizo, "Calculating the Squared Euclidean Distance for Vertical Data Represented in pTrees", in the Conference Proceedings of Software Engineering and Data Engineering (SEDE-2012), held on June 2012, Los Angeles, California, USA.

[6] M. Khan, Q. Ding, and W. Perrizo, K-Nearest Neighbor Classification of Spatial Data Streams using P-trees, Proceedings of the PAKDD, pp. 517-528, 2002.

[7] A. Perera, A. Denton, P. Kotala, W. Jockhec, W.V. Granda, and W. Perrizo, P-tree Classification of Yeast Gene Deletion Data. SIGKDD Explorations, 4(2), pp.108-109, 2002.

[8] I. Rahal and W. Perrizo, An Optimized Approach for KNN Text Categorization using pTrees. Proceedings of ACM Symposium on Applied Computing, pp. 613- 617, 2004**.**

[9] Morris Mano, Digital Design,3rd Edition, Prentice Hall, 2001.

[10] A.M. Law andW.D. Kelton. Simulation modeling and analysis. McGraw-Hill series in industrial engineering and management science.McGraw-Hill, 2000. ISBN 9780070592926.

[11] D.C. Montgomery. Design and analysis of experiments. Wiley, 1976. ISBN 9780471614210.