Generalized Search Algorithm For Algebraic Functions

Quanzheng Long[†]

[†]*Purdue University*

qlong@cs.purdue.edu

ABSTRACT

Multi-dimensional algebraic functions are often used to make abstract models for real-world data. One common problem is to search for data that has the smallest function value. Another problem is to search for data whose function values are in a given range. As the dataset is larger, linear search methods become increasingly more expensive. Existing techniques are only applied to some specific scenarios, or having some limitations. In this paper, we present an efficient search algorithm and data structure that are applicable to any algebraic function.

Categories and Subject Descriptors

H.3.4 [**Search**]: Index Query processing

1. INTRODUCTION

One of the common problems is to search for the data element with the smallest function value. Consider the following motivating applications.

Example 1: Determining a server location for clients. The server location is chosen from a limited set of candidate locations. The optimal server location has the minimum of summary distance to all the clients. When a company has only one client, the solution is the location nearest to that client. This problem is solved by a classic kNN algorithm, e.g., [4]. However, if a company has more than one clients, for an example of two clients, the problem is out of the scope of the classic kNN algorithm. The distance to minimize is

$\sqrt{(x-c_{11})^2+(y-c_{12})^2}+\sqrt{(x-c_{21})^2+(y-c_{22})^2}$,

where $< c_{11},\ c_{12} > and < c_{21},\ c_{22} >$ are the locations of the two client, $< x, y >$ is the location of a candidate server. Tailored Group Nearest Neighbor(GNN) algorithms [?] exist to specifically for this scenario.

Example 2: Searching for speeding vehicles in a given time interval and a given region. In this scenario, the input contains the locations and corresponding time for all vehicles. The objective speed function is:

$\frac{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}}{t_2-t_1}$,

where $< x_1,\ y_1 >$ is the location at Time t_1, and $< x_2,\ y_2 >$ is the location at Time t_2. The problem is to search for any vehicle whose objective function value is greater than the maximum speed limit.

Also, some data structures and algorithms e.g., the TB-Tree and STB-Tree[?], are already designed specifically for this scenario.

Example 3: Customized recommendation. Internet companies like Airbnb or Yelp provides searching for the most "relevant" items to their customers. The relevant value of an item is often calculated by certain algebraic functions which take care of lots of factors, such as distance to the customer and number of starred reviews. The weight value for each factor can be customized. Therefore, the objective function is:

$W_1 * \sqrt{(x-c_1)^2+(y-c_2)^2}+W_2 * z$,

where $< x,\ y >$ is the location of a candidate, z is its number of starred reviews, $< c_1,\ c_2 >$ is the location of a customer, and $W_1\ and\ W_2$ are the customized weight values for that customer.

The three application examples above demonstrate the need to search for different algebraic functions. Linearly searching through the dataset and applying the objective function for each data item is expensive. Indexing techniques, e.g., the M-tree [1], speeds up the search by avoiding this linear search. However, one limitation of the M-tree is that it applies only to the metric space and depends heavily on the triangular inequality.

This paper investigates the problem of searching for arbitrary algebraic functions, and proposes a unified algorithm that has overcome limitations of traditional search techniques. More specifically, we focus on efficient ways for answering the top-k and range query problems when using generic algebraic functions. In other words, distance functions in metric and non-metric spaces are both supported. Given the various objective functions, the target is to avoid as much as possible accessing data that does not contribute to the query results.

The results of the conducted extensive experiment demonstrate that the I/O requests are much less than linear searching technique.

The rest of this paper proceeds as follows. Section 2 formally defines the generalized algebraic function search problem, with some background material provided. Section 3 presents the search algorithm details for the problem, also analyzes the computational complexity. Section 4 proposes improved R*-tree for optimizing the search algorithm. Section 5 reports the extensive experiment results. Section 6 concludes the paper.

2. PRELIMINARIES

In this section, we formally define the algebraic search problem that we address in this paper. Also, we overview the spatial index and partial derivative that serve as background materials to facilitate explaining the proposed algorithm in Section 3.

2.1 Algebraic Function Search Problem

Consider an n-dimensional dataset S containing N vectors. For each vector, say $x \in S$, has n dimensions.

$$\text{x=} < x_1,\ ...,\ x_i, ...,\ x_n >, \\ \text{where i=1, ..., n.}$$

The algebraic function $f(x)$ is defined on x. As shown in the three motivating examples in the introduction, we replace the variables with vector definition.

$$f(x) = \sqrt{(x_1 - c_{11})^2 + (x_2 - c_{12})^2} + \sqrt{(x_1 - c_{21})^2 + (x_2 - c_{22})^2}$$

$$f(x) \frac{\sqrt{(x_2 - x_1)^2 + (4_2 - x_3)^2}}{x_6 - x_5}$$
$$f(x) = W_1 * \sqrt{(x_1 - c_1)^2 + (x_2 - c_2)^2} + W_2 * x_3$$

The search problem is described based on algebraic function. In this section, we only define the k-smallest and range searches for simplicity. There are other search types, such k-nearest search, which are discussed briefly at the end of Section 3.

DEFINITION 1. ***(k-smallest search)*** *Given an integer number k, search for vectors which have the k smallest function values of f(x). Formally, the search result denoted as KS(S,f(x),k), is a set of vectors that* $\forall o \in KS(S, f(x), k), \forall s \in S - KS(S, f(x), k), f(o) \leq f(s)$

DEFINITION 2. ***(range search)*** *Given two values LEFT and RIGHT, where LEFT* $\leq$ *RIGHT. Search for all vectors whose function values are between LEFT and RIGHT. Formally, the search result, denoted as RT(S, f(x), LEFT, RIGHT), is a set of vectors that* $\forall o \in RT(S, f(x), LEFT, RIGHT), LEFT \leq f(o) \leq RIGHT$

2.2 Partial Derivative

To investigate properties of a generic algebraic function, its partial derivatives are used. Intuitively, a partial derivative represents how fast the function value increases (when the derivative is positive) or decreases (when the derivative is negative) on that dimension.

Formally, the derivative to dimension x_i is denoted as f'_{x_i}.

$$f'_{x_i} = \lim_{h->0} \frac{f(x_1, ..., x_i + h, ..., x_n) - f(x_1, ..., x_i, ..., x_n)}{h}, \\ for\ 1 \leq i \leq n$$

COROLLARY 1 When $f'_{x_i} \geq 0$, f(x) is not decreasing as x_i increases; when $f'_{x_i} \leq 0$, f(x) is not increasing as x_i decreases.

Formally, given $f'_x \geq 0\ \forall x \in [MIN,\ MAX],\ if\ MIN \leq x_i \leq x_j \leq MAX,\ then\ f(x_i) \leq f(x_j)$. And vise versa.

For example, given the partial derivatives of the objective function in the motivating example 1.

$$f(x) = \sqrt{(x_1 - c_{11})^2 + (x_2 - c_{12})^2} + \sqrt{(x_1 - c_{21})^2 + (x_2 - c_{22})^2}$$
$$f'_{x_1} = \frac{x_1 - c_{11}}{\sqrt{(x_1 - c_{11})^2 + (x_2 - c_{12})^2}} + \frac{x_1 - c_{21}}{\sqrt{(x_1 - c_{21})^2 + (x_2 - c_{22})^2}}$$
$$f'_{x_2} = \frac{x_2 - c_{12}}{\sqrt{(x_1 - c_{11})^2 + (x_2 - c_{12})^2}} + \frac{x_2 - c_{22}}{\sqrt{(x_1 - c_{21})^2 + (x_2 - c_{22})^2}}$$

Matured mathematical algorithms and systems, such as Maxima[5], is able to compute the partial derivative formulas of a given function. Using the partial derivative formula, we then calculate the **value ranges** of derivatives in a postfix notation(Reverse Polish notation)[2]. These procedures are discussed in the next section.

2.3 Value Range

DEFINITION 3. ***(value range)*** *The value range is a group of pairs of vmin and vmax, where* $vmin \leq vmax$. *A value range of a derivative means the derivative value is between one of the pairs.*

Formally a value range is denoted as VR, $VR = \{[vmin_1, vmax_1],\ ...,\ [vmin_t, vmax_t]\}$ *where* $vmin_i \leq vmax_i\ for\ 1 \leq i \leq t$. *The value range of* f'_{x_i} *means that* $\exists 1 \leq j \leq t, vmin_j \leq f'_{x_i} \leq vmax_j$.

We design a procedure to calculate the value range of derivatives. Given a mathematical expression of a partial derivative, and the value ranges of its variable, the procedure is to use the postfix notation[2].

The procedure is similar to calculating a exact value of a mathematical expression given exact variable values. However, the mathematical operators take different behaviors when processing value range calculation.

Therefore, we redefine the mathematical operators for value range calculation. Only the procedures of addition and multiplication operators (+ and *) are presented for simplicity, as shown in Algorithm 1. Other operators, such as division or sine, are not difficult to design.

Algorithm 1 Operations for value range

```
 1: procedure PLUS(VR1,VR2)
 2:     initialize VR
 3:     for pair( vmini, vmaxi) in VR1 do
 4:         for pair( vminj, vmaxj ) in VR2 do
 5:             VR.add( vmini+ vminj, vmaxi+ vmaxj)
 6:         end for
 7:     end for
 8:     return VR
 9: end procedure
10: procedure MULTIPLE(VR1,VR2)
11:     initialize VR
12:     for pair( vmini, vmaxi) in VR1 do
13:         for pair( vminj, vmaxj ) in VR2 do
14:             vmin = min ( vmini * vminj, vmini * vmaxj,
                vmaxi * vminj, vmaxi * vmaxj )
15:             vmax = max ( vmini * vminj, vmini * vmaxj,
                vmaxi * vminj, vmaxi * vmaxj )
16:             VR.add( (vmin, vmax) )
17:         end for
18:     end for
19:     return VR
20: end procedure
```

Processing Example:
Calculate derivative $f'x_1 = x_1 * (x_1 + x_2)$.
Initially $VR(x_1) = \{(-1, 2)\}$, $VR(x_2) = \{(-1, 2)\}$.
Then $VR(x_1 + x_2) = \{(-2, 4)\}$.
Finally $VR(f'x_1) = \{(-4, 8)\}$

Note that during the calculation, the value range might be amplified. In other words, the value range calculated becomes wider than the value of expression actually is. This is expected and does not affect our algorithm very much.

To avoid the amplification, sometimes a formula may be simplified. Since our algorithm only cares whether or not the value range is non-negative or non-positive. For example of the Euclidean distance function, f(x)=$\sqrt{x_1^2+x_2^2}$, one the derivatives is $f'_{x_1} = \frac{x}{\sqrt{x_1^2+x_2^2}}$. However $g'_{x_1} = x$ is used as an equivalent because $\sqrt{x_1^2+x_2^2} >= 0$.

2.4 Multi-dimensional Index And MBR

To avoid accessing original dataset when searching, we try to take fully use of multi-dimensional indexes. A lot of spatial indexes, such as R-Tree[3] and Quad-Tree[**?**], are suitable for our algorithm. Any multi-dimensional index or spatial index is able to be applied to our algorithm, as long as it has the following features.

1) Group vectors together into **MBRs**, when those vectors are closed to each other;

2) An MBR is stored in an index node, with other meta information and links to other index nodes.

3) Build a top-down tree architecture that an upper node points to the the bottom ones.

Most spatial indexes are developed on top of MBR (Minimum Bounding Rectangle). We are using the same definition as in [**?**].

DEFINITION 4. *(**MBR**)An MBR is defined by two end points, S and T, of the rectangle's major diagonal.*
$MBR = [S,T], where$
$S =< s_1, ..., s_i, ..., s_n >,$
$T =< t_1, ..., t_i, ..., t_n >,$
$\forall x =< x_1, ..., x_i, ..., x_n >$ *under the node,*
$s_i \le x_i \le t_i$ *for* $1 \le i \le n$.
An MBR has a minimum property that:
$\forall \epsilon > 0, \forall 1 \le i \le n, MBR' = [S,T'], T =< t_1, ..., t_i - \epsilon, ..., t_n >, \exists x =< x_1, ..., x_i, ..., x_n >$ *under the nodethatx* $> T'$.
Also, $\forall \epsilon > 0, \forall 1 \le i \le n, MBR' = [S',T], S' =< s_1, ..., s_i + \epsilon, ..., s_n >, \exists x =< x_1, ..., x_i, ..., x_n >$ *under the nodethatx* $< S'$.

There are four types of index nodes in a tree structured index:

1) An **Entry node** stores the original data of a row (including vectors and unindexed data). An entry node also has an MBR even there is only one vector in it. Both the two endpoints of the MBR are the exactly the vector stored in the node.

2) A **Leaf node** stores its MBR and the links to the entry nodes belong to it.

3) An **Internal node** stores its MBR and the links to its children node(internal nodes or leaf nodes).

4) A **Root node** is a special internal node that provides a start of accessing.

Figure 1 is an example of multi-dimensional index with a tree structure.

DEFINITION 5. *(**monotones**) An index node is **monotonic** for f(x), if and only if its MBR is monotonic in all dimension.*

*For each dimension i of vector x, MBR=[S, T] is **monotonic**, if and only if any* $1 \le i \le n, S \le x \le T$*, f(x) is derivable to* x_i*, and* $VR(f'_{x_i}) \le 0 or \ge 0$.

Definition 5 is the principal component of this paper. When an index node is monotonic, the algorithm can calculate the metrics without knowing any vectors or its children nodes. We will discuss the details in the next section.

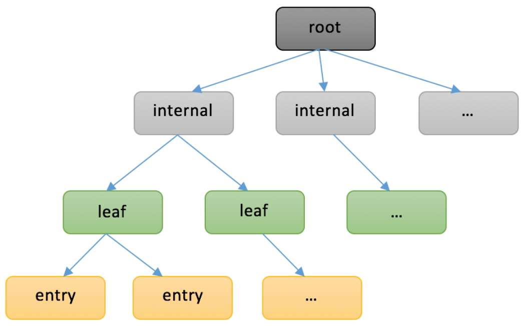

Figure 1: A tree structured multi-dimensional index

3. SEARCH ALGORITHM

In this section, we present details of k smallest search algorithm. The k-nearest and range search algorithms are also discussed briefly at the end of the section.

Firstly the metrics to prune unnecessary branches is presented, then followed by the pseudo code of the algorithm. The metrics are based on the preliminaries given above.

3.1 Metrics

In previous section, we mentioned that an MBR represents the domain of all data vectors under the node. To avoid accessing the original data vectors(entry nodes), two metrics of the node, MINVAL and MINMAXVAL, are computed.

DEFINITION 6. *(**MINVAL**) The MINVAL(minimum value) of an index node nd, denoted as MINVAL(nd,f(x)),*

$$\begin{cases} f(p), & entrynode; \\ min(f(q)), & monotonic; \\ min(MINVAL(ch, f(x)) & otherwise; \end{cases} \quad (1)$$

*where p is the data vector in the entry node, q is any **MBR corner**, and ch is any child node of nd.*

DEFINITION 7. *(**MBR corners**) The corners of an MBR=(S,T) is a set of vectors,* $\{x| < x_1, ..., x_i, ..., x_n > \}$*, where* $1 \le i \le n$*,* $x_i = s_i$ *or* t_i

When nd is an entry node, MINVAL is the function value. When the MBR of nd is monotonic for f(x), MINVAL is the minimum of all the **corners of the MBR**. If it is neither of the two cases, then calculate the minimum of all the child nodes' MINVALs.

THEOREM 1 The function value of any vector under the node nd is greater than or equal to MINVAL(nd, f(x)).

PROOF:
1) If nd is an entry node, there is only one vector p under it, hence, the function value of any data vector under nd is equal to f(p).

2) According to Definition 5, when nd is monotonic for f(x), it means that f(x) is consistently increasing or decreasing over the MBR when x_i increases for $\forall 1 \le i \le n$. Therefore, the minimum of all MBR corners' function values is smaller than or equal to the function value at any point in the MBR. The minimum of all MBR corners' function values is MINVAL.

3) When the node is not monotonic, the MINVAL is a minimum of all MINVALs of the child nodes. Hence, the function value of any data vector under nd is still greater than or equal to the MINVAL.

An MINVAL of a node is the lower bound of the all vectors under the node. MINVAL guarantees that every vector is greater or equal to it. Similarly, **MINMAXVAL** serves as an upper bound which guarantees that there exists at least one vector of which function value is smaller or equal to MINMAXVAL.

The basic idea of construction is to use the property of MBR. Imagine the faces of an MBR, and for each face, there exists at least one vector inside the face–otherwise we can get a smaller MBR. Therefore, when a non-entry node is monotonic for f(x), our algorithm calculates the maximum values of all faces, and the minimum of the maximum values is MINMAXVAL.

DEFINITION 8. ***(MINMAXVAL)*** *The MINMAXVAL(mini-maximum value) of an index node nd, denoted as MINMAXVAL(nd,f(x)),*

$$\begin{cases} f(p), & entrynode; \\ min(max(f(corners\ of\ \textbf{MBR faces})), & monotonic; \\ min(MINMAXVAL(ch, f(x)) & otherwise; \end{cases} \quad (2)$$

where p is the vector in the entry node, and ch is any child node of nd.

DEFINITION 9. ***(MBR faces)*** *A MBR face is an (n-1) dimensional MBR. There are 2*n faces. Each face has* 2^{n-1} *corners, because there are* 2^{n-1} *combinations of choosing S or T.*

Formally, for the j^{th} *corner of* i^{th} *face, corner* $q =< q_1, ..., q_k, ...q_n >$, *where* $q_k=$

$$\begin{cases} s_k, & k = (i+1)/2\ AND\ i\ mod\ 2 = 1; \\ t_k, & k = (i+1)/2\ AND\ i\ mod\ 2 = 0; \\ s_k\ or\ t_k, & depends\ the\ \textbf{MBR corners}; \end{cases} \quad (3)$$

where $1 \leq i \leq 2 * n, 1 \leq j \leq 2^{n-1}, 1 \leq k \leq n.$

THEOREM 2 There exists at least one vector under node nd, whose function value is less than or equal to MINMAXVAL(nd,f(x)).

PROOF:

1) If nd is an entry node, there is only one data tuple p under it. Hence, the function value of any data tuple under it is equal to f(p).

2) According to the minimum property of MBR in Definition 4, there exists at least one vector on each face. Also according to Definition 5, when nd is monotonic for f(x), it means that f(x) is consistently increasing or decreasing over any MBR faces. Therefore, for any face, there exist an vector whose function value is less than or equal to the maximum of MBR face corners' function values. MINMAXVAL is chosen from one of the MBR faces hence there is always existing an vector whose the function value is less than or equal to MINMAXVAL.

3) When the node is not monotonic, the MINMAXVAL is a minimum of all MINMAXVALs of the child nodes. Hence, there exists at least one vector under nd with the function value less than or equal to the MINMAXVAL.

Note that there is a more efficient way to calculate MINMAXVAL than the way of definition. It is more efficient because it avoids iterating all possible corners for each face. There are 2^n corners for an MBR, however, for each face there are half of the MBR's corners(2^{n-1}). To obtain the maximum value of corners for each face, sort out all the corners' function values in increasing order. Finally for each face, iterate in that order and choose the first corner that belongs to that face. The function value of that first corner is the maximum value of corners for the face.

THEOREM 3 Given two index nodes nd_1 and nd_2, if MINVAL(nd_1,f(x))>MINMAXVAL(nd_2,f(x)), then the vector with the smallest function value must be under node nd_2.

3.2 Pseudo Code

In this section, we present the pseudo code of the k-smallest search algorithm. We assume that the dataset has at least k vectors so that edge checking is ignored.

As Algorithm 2, it has two main steps to get the top-k tuples: to search for smallest value, then search for the remaining smallest values.

Algorithm 2 k-smallest-search

```
1: procedure KSmallestSearch($k, root, f(x), T$)
2:     $initlstStk, minStk, actLst$
3:     $actLst.put(root)$
4:     $SmallestSearch(k, f(x), T, actLst, minStk, lstStk)$
5:     $RemainingSearch(k, f(x), T, actLst, minStk, lstStk)$
6:     return $actLst$
7: end procedure
```

Initially, an active list is initialized for R-Tree nodes. The root node is put into the active list. Then recursively execute the following process until reaching entry nodes.

1) Retrieve all child nodes of nodes in the active list;

2) Visit the child nodes and compute their MINVALs and MINMAXVALs;

3) Based on the metrices and Theorem 3, prune some nodes to avoid further access.

4) Since the pruned nodes may be used later, they are pushed into a stack.

The details are shown in Algorithm 3.

At the end of the first step, there might be more than one vector that have the same smallest function value. Assume m vectors, where $1 \leq m \leq k$.

In the second step, the algorithm looks for the remaining k-m vector with the smallest function values. The pruned nodes are visited one by one, in order of popping from the stack, until obtaining the remaining k-m vectors. We skip the pseudo code for simplicity. It is similar to Algorithm 3, and not difficult to implement.

This algorithm exploit the usage of index. The key idea is to calculate MINVALs and MINMAXVALs for all index nodes visited, only using their MBRs.

Algorithm 3 smallest-search

```
1: procedure SmallestSearch($k, f(x), T, actLst, minStk, lstStk$)
2:     while $NonEntry(actLst.first())$ do
3:         $inittmpLst$
4:         for $node\ n\ in\ actLst$ do
5:             for $cn\ in\ node.children$ do
6:                 $tmpLst.add(cn)$
7:             end for
8:         end for
9:         $actLst.clear()$
10:        min_MINMAX = min ( MINMAXVAL ( tmpLst.nodes, f(x), T ) )
11:        for $n\ in\ tmpLst$ do
12:            if min_MINMAX >= MINVAL ( n, f(x), T ) then
13:                actLst.add(n)
14:                tmpLst.del(n)
15:            end if
16:        end for
17:        if tmpLst.size()>0) then
18:            min_MIN = min ( MINVAL ( tmpLst.nodes, f(x), T ) )
19:            minStk.push(min_MIN)
20:            lstStk.push(tmpLst)
21:        end if
22:    end while
23: end procedure
```

3.3 Other Search Query Types

Other types of algebraic function searches, such as range and k-nearest searches are also supported.

Range search is much more straightforward than k-smallest search. In addition to MINVAL, MAXVAL is defined similarly. It guarantees that every function value of vectors in the nodes is less than or equal to MAXVAL. The result to range search query is also retrieved by visiting nodes in active list from top to bottom. In the meantime of visiting, prune nodes that are out of the queried range until all nodes are in the range.

DEFINITION 10. ***(k-nearest search)*** *Given a target value T and an integer number k, search for the k tuples out of a dataset DS, with the function values of f(x) nearest to T. Formally, the search result denoted as KN(DS,f(x),T,k), is a set of tuples that* $\forall o \in KN(DS, f(x), T, k), \forall s \in DS - KN(DS, f(x), T, k), |f(o) - T| \leq |f(s) - T|$

THEOREM 4 A k-nearest search problem can be converted into a k-smallest search problem.

PROOF Construct another function g(x)=|f(x)-T|, then the k smallest search for g(x) is equivalent to the k nearest search for f(x) with the target value T.

Theorem 4 shows that we can use the k smallest search algorithm to solve the k nearest search problem. However, a function expression in a form of $g(x) = |f(x)-T|$ is not easy and efficient to apply the solution. Because when computing the metrics for an index node, the algorithm requires the formulas of partial derivatives. Since $g(x) = |f(x) - T| = \sqrt{(f(x) - T)^2}$, the partial derivatives obtained through this formula might be very complex.

To overcome this difficulty, two different metrics for the k nearest search, MINDIST and MINMAXDIST can be designed to replace MINVAL and MINMAXVAL. MINDIST is the lower bound to replace MINVAL, and MINMAXDIST is the upper bound in place of MINMAXVAL. They are more efficient to compute than MINVAL and MINMAXVAL. The definitions are trivial and can be written similarly.

3.4 Algorithm Performance

The algorithm complexity depends on the quality of the index structure and the nature of the algebraic function applied. In best case, the algorithm would exploit the usage of index, visiting very few internal nodes to locate the target entry nodes. However in worst case, the algorithm would iterate every nodes as linear scanning.

When a function is not derivable, or the MBRs are not monotonic, the algorithm performs closed to the worst case. For any Weierstrass function[6], the algorithm always performs at the worst case because this type of function is not derivative at all.

Most algebraic functions in real life are derivable. However, some algebraic function, such as application example 2, performs badly in some dataset even the function is derivable. Because whether the MBRs are monotonic or not heavily depends on the dataset used, and the multi-dimensional index applied.

One of the well known data structure for multi-dimensional data is R*-tree. It provides balanced structure and splitting rectangles with very little overlap. It performs well for lots of common algebraic functions. However there are still some potential improvements.

In next section, we investigate the bad performace issue of the R*-Tree and propose improvements on the R*-Tree to make it works better for some difficult case like speed function.

4. IMPROVED R*-TREE

R*-Tree[**?**] is one of most popular data structure for multi-dimensional data. It is developed from R-Tree[3] as a member of the R-Tree family.

The original R-Tree is an natural extension of B-Tree. Similarly, it provides insert/delete/search operations. When inserting a new item to a R-Tree, choosing a node to insert and how to split a full node are two critical problems. The two problems lead researchers to develop different members of R-Tree family. Among those members, R*-Tree is proved to be one of the best members providing good quality of choosing and splitting nodes.

4.1 Root Causes of Bad Performance

When applying R*-Tree to our algorithm, most of the algebraic functions perform well. The algorithm can save at least 80% of disk page accessing in most cases. However, some kind of algebraic functions are not able to avoid too many disk accessing as expected. Such as the speed function in motivating application example 2:

$f(x) = \frac{\sqrt{(x_1-x_2)^2+(x_3-x_4)^2}}{x_5-x_6}$.

One of the root causes is $\frac{1}{x_5-x_6}$. That leads the algorithms of choosing inserting nodes and splitting nodes perform badly.

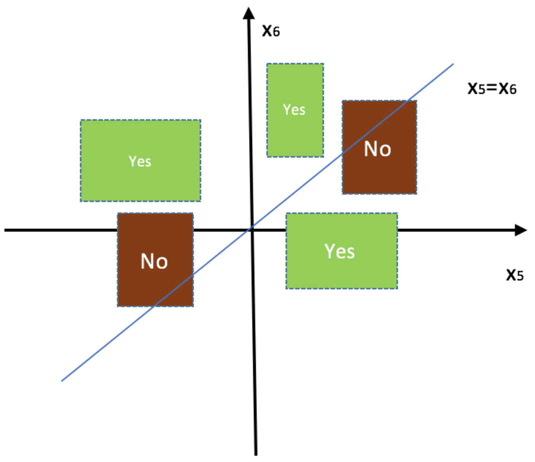

Figure 2: an MBR is better not across line $t_2 = t_1$

Assume that
$t_{11} <= x_6 <= t_{12}$,
$t_{21} <= x_5 <= t_{22}$,
$t_{01} <= x_5 - x_6 <= t_{02}$,
What if $t_{01} < 0$ and $t_{02} > 0$? Then $\frac{1}{x_5 - x_6}$ reaches $+\inf$ and $-\inf$. Hence the MBR with this value range is not monotonic.

To avoid this from happening, we conduct the following.
Let $t_{01} <= 0, t_{02} <= 0$ or $t_{01} >= 0, t_{02} >= 0$, then
$t_{12} >= t_{11} >= t_{22} >= t_{21}$ or
$t_{22} >= t_{21} >= t_{12} >= t_{11}$.

Intuitively, an MBR is better not across line $x_5 = x_6$ as shown in Figure 2.

The R*-Tree and other implementation of R-Trees, do not take care the factor that whether or not a MBR should across a line. Consequently, lots of MBRs are not monotonic.

Another root cause is that, every time splitting a R*-Tree node, it only affect one dimension. As more dimensions are used, it is less possible that the algorithm would split one specific critical axis. However, some axis's might be more promising to make a monotonic MBR, than other axis's.

To address those issues, we propose two improvements on R*-Tree. One improvement is on choosing subtree to insert new items, and another is on choosing split axis and index.

The final root cause we discover is that some algebraic functions have stronger locality than some others. A function with weak locality may require some distribution in the dataset to have very good performance. A strong locality means that the vectors closed to each other always have some closed function values, and the vectors far way may have very different function values.

For example, we consider $f(x) = x_1 * x_2$ has stronger locality than $f(x) = sin(x_1 * x_2)$. Therefore, algebraic function $f(x) = sin(x_1 * x_2)$ only has good performance when the dataset is very dense. Instead $f(x) = sin(x_1 * x_2)$ doesn't have that request.

4.2 Choosing Subtree

When choosing a subtree for inserting a new item, the original R*-Tree[?] takes into account factors including overlap enlargement, area enlargement and rectangle area. Those factors are beneficial for choosing a subtree that leads to

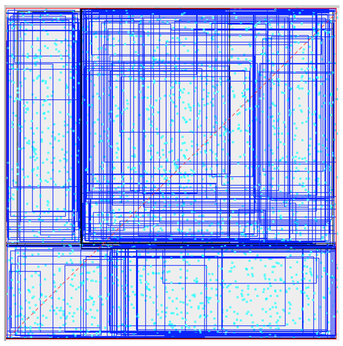

Figure 3: The original R*-Tree in x_5(horizontal) and x_6(vertical) dimensions. Many rectangles cross the dash red line $x_5 = x_6$

.

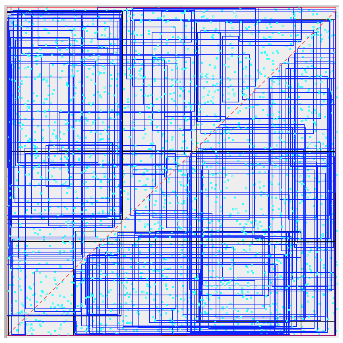

Figure 4: The improved R*-Tree has few rectangles across line $x_5 = x_6$, which leaves an empty line.

little overlap and area enlargement. However, they might also lead a index node changing from monotonic to non-monotonic for certain functions.

Therefore we propose that choosing subtree also needs to consider monotones change as an extra factor.

After inserting an item to a subtree node, the subtree node's MBR might have three cases of changing monotones:
1) Change from non-monotonic to monotonic;
2) Keep being monotonic;
3) Keep being non-monotonic;
4) Change from monotonic to non-monotonic;

Above is the preferred order for choosing a subtee. 1) > 2) > 3) > 4). The improving algorithm would choose the subtree in that above order. Then resolve the tier using the original R*-Tree's algorithm.

4.3 Choosing Split Axis and Index

When splitting a index node, a splitting axis and a splitting index need to be chosen. The original R*-Tree algorithm has a issue of causing some MBR changing from monotonic to non-monotonic.

Similar to the improvement on choosing subtree for inserting, we propose that the monotones is also considered. For each splitting axis and index, the number of child nodes that are monotonic is recorded. Then choosing the axis and index having the maximum number. Also, we resolve the tier using the original R*-Tree algorithm.

4.4 Visualized Comparison

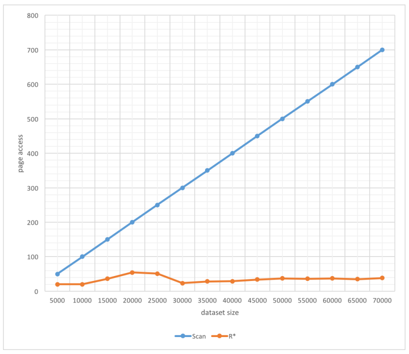

Figure 5: Page access for customized recommendation problem via different algorithms

We conduct an experiment to visually compare the original R*-Tree to the improved R*-Tree. The experiment uses the speed monitoring function, which has six variables (dimensions). 3000 randomly generated elements are inserted into both trees with the same order.

The results are shown in Figure 3 and Figure 4. We can intuitively observe the rectangles in dimension x_5 and x_6. The original R*-Tree put lots of rectangles across the line $x_5 = x_6$, which let them be non-monotonic. On the other hand, the improved R*-Tree avoid doing that as much as possible, hence Figure 4 has an empty line along $x_5 = x_6$.

The visual results give us confidence that the improved R*-Tree can work better than the original one. Consequently, we conduct benchmark tests for comparing their performance.

5. PERFORMANCE EXPERIMENTS

In this section, we study the performance of the proposed data structure and algorithm. Different algebraic functions are applied in different dataset. The number of disk page access is used as the performance metric.

5.1 Customized Recommendation

Algebraic function:
$f(x) = W_1 * \sqrt{(x_1 - c_1)^2 + (x_2 - c_2)^2} + W_2 * x_3$

Dataset:
In this case we use the yelp academic data**??** as business locations. Each business location contains a pair of longitude and latitude, and also the number of reviews serving as x_3. We manually set up different $< c_1, c_2 >$ as customer locations, and the weight values for distance and factor x_3. The customer location should not be very far away from all candidates. As long as the location is reasonable, the algorithm is always efficient.

Result:
In Figure 5, the page access grows much slower as the dataset size increases, when using the new algorithm(yellow curve), compared to linear scanning technique(blue curve).

5.2 Server Location

Algebraic function:
$f(x) = \sqrt{(x_1 - c_{11})^2 + (x_2 - c_{12})^2} + \sqrt{(x_1 - c_{21})^2 + (x_2 - c_{22})^2}$

Dataset:
In this case we still use Yelp academic data[**?**] as candidate locations for server. Since it is the exact same dataset, we can use the exact same index built for Section 5.1.

Also, we manually set up two locations as clients. Those constant can also affect the performance. The two client locations should not be very far away from all candidates. As long as the locations are reasonable, the algorithm is always efficient. Also the number of clients can be vary.

Result:
In Figure 6, the page access grows significantly slower using the proposed technique(yellow curve), compared to linear scanning technique(blue curve), as the dataset size increases.

Note that we are using the exact same dataset and index as we use for server location problem. The index is built with more dimensions (x_1, x_2, x_3) than the dimensions that are queried(x_1 *and* x_2). This suggests that one index can be reused in different algebraic functions. It exploits the usage of index and enables more functionality of database system.

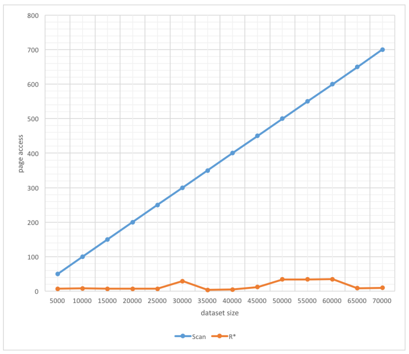

Figure 6: Page access for server location problem via different algorithms

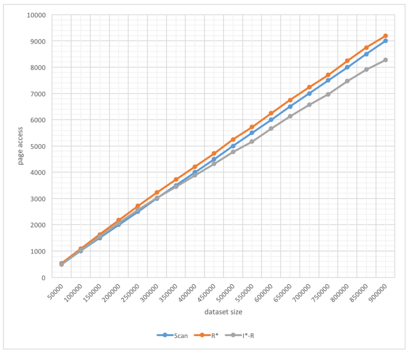

Figure 7: Page access for speed monitoring problem via different algorithms

5.3 Speed Monitoring

Algebraic function:
$f(x) = \frac{\sqrt{(x_3 - x_1)^2 + (x_4 - x_2)^2}}{x_6 - x_5}$

Dataset 1:
In this case we use BerlinMOD data[**?**] for vehicles trips. The dataset contains the start and end of locations and time.

Result:
In Figure 7, the page access grows mostly the same via different techniques. However the improved R*-Tree(gray curve) works better than original R*-Tree(yellow curve) and linear scanning(blue curve).

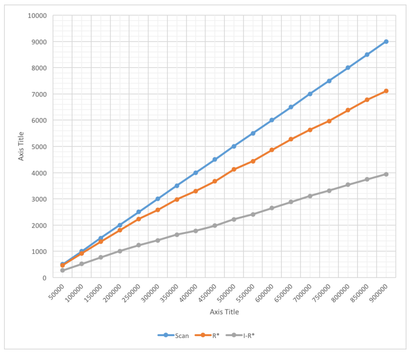

Figure 8: Page access for speed monitoring problem via different algorithms using random start and end time

Section 4.1 discussed the root causes of the bad performance. One of the cause the nature of the algebraic function. This function has very week locality.

Another cause is the dataset distribution. The speeds are mostly in a small range and in a particular pattern. The start and end time are in a sequential order, which would make the rectangles too large. We verify this cause using a different dataset.

Dataset 2:

In this dataset, we still use BerlinMon, however we change the start and end time. Instead, we use the values of x_1 and x_4 and start and end times. This would break the x_5 and x_6 into more random data.

Result:

In Figure 8, our proposed algorithm with improved R*-Tree performs better than with original R*-Tree. And they both work better than linear scanning. This verify that the data distribution can affect the performance.

6. CONCLUSION

In this paper, we presented the new algorithm for generic algebraic function search. We analyzed the performance and complexity of the algorithm and proposed improvement on data structure. Our extensive experimental results shows the efficiency and availability of the techniques.

Essentially, our new technique is a natural extension of spatial index and classic kNN algorithm. This extension is enabling the database indexing to be more powerful in real-world problems.

7. REFERENCES

[1] M-tree an efficient access method for similarity search in metric spaces. www.cidrdb.org, 1997.

[2] D. W. W. Arthur W. Burks and J. B. Wright. An analysis of a logical machine using parenthesis-free notation. *American Mathematical Society*, 8(64):53–57, Apr. 1954.

[3] A. Guttman. R-trees: A dynamic index structure for spatial searching. SIGMOD, 1984.

[4] N. Roussopoulos. Nearest neighbor queries. *SIGMOD*, 8(64):53–57, Feb. 1995.

[5] K. Velten. *Mathematical Modeling and Simulation: Introduction for Scientists and Engineers*. Wiley, 2009.

[6] K. Weierstrass. Abhandlungen aus der functionenlehre [treatises from the theory of functions]. page 97, 1886.