REDUNDANCY ON CONTENT-BASED INDEXING

By

CHEUNG King Lum Kingly

A Thesis

submitted in partial fulfillment of the requirements

for the Degree of Master of Philosophy

Division of Computer Science and Engineering

The Chinese University of Hong Hong

1997

Abstract

Multimedia is so popular that a lot of multimedia applications have been created. Those applications usually deal with huge amount of data and it is necessary for them to have an indexing structure such that efficient retrieval of data can be provided. However, traditional indexing techniques cannot provide a satisfactory performance on searching, especially when they are dealing with high dimensional data which is very commonly found in those applications. R-Tree, which is one of the most efficient content-based indexing structure, is still not efficient enough on high dimension data retrieval. The major problem of R-Tree is due to its non-deterministic behavior during searching. We, therefore, propose a new index structure, which is Redundant Tree, as well as its searching algorithms so that a better performance on searching can be obtained. The basic idea of the Redundant Tree is that we introduce redundancy on R-Tree. It inherits advantages from R-Tree, and it eliminates disadvantages by its own properties. Furthermore, we propose exact searching and nearest neighbor searching algorithms for Redundant Tree so that non-deterministic searching behavior is minimized or even eliminated. We present the results from a series of experiments which show that our Redundant Tree outperforms R-Tree on both exact and nearest neighbor search.

Acknowledgement

I would like to express my sincere thanks to Prof. Ada Fu and Prof. K. S. Leung, my supervisors of this research, for their advice and encouragement. In the early drafts of this thesis, they found many errors and gave me valuable suggestions and corrections. Also, I want to thank Prof. M. H. Wong and Prof. C. Lu for their comments that contributed to the completion of this thesis.

Furthermore, the Department of Computer Science and Engineering of the Chinese University of Hong Kong has provided an ideal working environment. I am grateful to all staff in the department for providing assistance. Colleagues have also provided activities for relax. Thanks to T. M. Wong, Chris Cheung, Mary Leung, Paul Law, Alan Tung, Peter Lie and Y. H. Hung.

Additional technical assistance was provided by many individuals. I especially thank Polly Chan for her valuable information on my research. This thesis was typeset with LaTeX. Arthur Hsieh, Terence Wong and S. K. Lam were my informants for LaTeX. There were many problems of submitting the thesis. Thanks to C. M. Kuok for his help.

Thanks to my father–Frank Cheung, my mother–W. F. Lau and my sister–Queenie Cheung for their endless support. The love and patience of my family made this research possible. Finally, I thank T. Y. Law for her love and support during the writing of this thesis. I dedicate this thesis to her.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

As multi-media is being developed, images are generated at strictly increasing rate by a lot of applications. The development brings a lot of impacts on computer science. For example, the traditional database can deal with text data by exact information retrieval. However, it is not enough when image data are being retrieved. Image data, on the other hand, is quite different from the original text data. Besides exact-match retrieval, similarity retrieval and nearest neighbor retrieval algorithm has to be devised for image applications. The size of images are relatively large when they are compared with the text data. If the number of images in the database is very large, efficiency in accessing data becomes very important. The size of image data is very large and the dimension of image data is high also. Traditional indexing methods seems not to be appropriate or even not able to deal with large amount of high dimension data. Hence, we need a new indexing technique for multi-dimension data.

A lot of indexing techniques are developed to satisfy the needs. However, each of them has disadvantages. For example, R-tree and its variants are some of the most commonly used indexing structures in multi-media databases. However, their performance on searching is not good enough to deal with high dimension image data, and their performance is even worse in nearest neighbor searching. We want to ask, if there is any method that can help improve the indexing techniques? In order to answer the question, first of all, we have to know why R-Tree algorithm have such deficiency. And then we can provide solution to the identified problem.

1.2 Problems in Content-Based Indexing

Existing content-based indexing techniques cannot provide satisfactory performance on data retrieval. No matter whether it is exact or nearest neighbor search, a lot of nodes have to be accessed in order to get the desired object. R-Tree and R*-Tree are the most commonly used content-based index structures. However, they still have the problem. Furthermore, the performance on nearest neighbor search is much worse than exact search. Besides the fact that finding the nearest neighbor does need more steps, additional metrics to find the nearest neighbor may slow down the processing speed.

We propose solution to the addressed problem. In this thesis, we formulate the causes of these problems. First of all, we propose the use of redundancy on content-based indexing so that a better performance on searching can achieved. Searching algorithms of those Redundant trees have been proposed and experiments for performance evaluation of several content-based index tree have also been conducted. In addition, we discover a new nearest neighbor search algorithm that can give better performance.

1.3 Contributions

This thesis focuses on index structure and searching techniques on content-based indexing and presents new indexing structures as well as new searching algorithms to improve data retrieval performance. The main contributions of this thesis include the following research results:

1. It has been discovered that it is not necessary to use $MINMAXDIST$ in nearest search. A new pruning heuristic for nearest neighbor search has been derived so that we try to eliminate the metric from the pruning heuristics used in nearest neighbor search algorithm. The heuristic is proved to be correct. The new heuristics have been shown that they can replace the old pruning heuristics in [6]. Since we have proved the pruning heuristics are redundant, based on the newly derived heuristic an improved nearest neighbor search algorithm has been designed. The algorithm uses the new pruning heuristic which does not use a computation expensive metric. Furthermore, a defect of the original nearest neighbor has been pointed out, and its solution has also been provided. In addition, a N-nearest neighbor search algorithm has also be provided. Experiments show that the new nearest neighbor search algorithm uses less CPU time for processing while it does not introduce any extra number of node access. Therefore, the improve nearest neighbor search algorithm provides a better performance of data retrieval.

2. The feature of overlapping nodes in R-Tree has been described. The feature induces backtracking which makes R-Tree and its variant give a bad performance on searching. A new content-based index structure, Redundant R-Tree, has been designed and implemented to solve the problem cause by the presence of overlapping node. The exact search and neighbor search algorithm of Redundant Tree has also been provided to help the

tree give a better performance. Experiments have been done to show that our proposed index structure and searching algorithms outperforms the others such as R-Tree and R*-Tree.

1.4 Thesis Organization

In this thesis, we will focus on the bad performance of existing content-based index structures on data retrieval, and propose new index structure and search algorithms to solve the addressed problem. An introduction will be given in Chapter 1. In Chapters 2 and 3, we will briefly describe several content-based index structures and their searching algorithms. A problem that makes the nearest neighbor search on R-Tree become inefficient will be discussed, and an improved nearest neighbor search algorithm will then be presented in Chapter 4. In Chapter 5, we will describe the feature of overlapping nodes in R-Tree and how it prohibits efficient searching. In Chapter 6, we introduce our proposed content-based index structure Redundant Tree to solve problems that existed in R-Tree. Both exact search and nearest neighbor search algorithms of Redundant Tree and examples of searching will be given in Chapter 7. Furthermore, experimental results will be presented in Chapter 8, and the results show our proposed index structure outperforms the originals. In Chapter 9, we will make a conclusion and describe possible future research.

Chapter 2

Content-Based Indexing Structures

Content-based index structures aim at providing methods for retrieving multi-dimensional image data based on the images' contents. As multi-media applications are being developed, many content-based indexing structures are developed. Usually, the bounding region of a node will be represented as rectangles or hyper-rectangles because of its simplicity. The most commonly used strategy of handling rectangles is to divide the original space into appropriate sub-regions, distributing the children of the original space into the sub-regions. The main difference between different content-based index structures are how they insert data into and delete data from trees, and when and how they split nodes.

In this Chapter, several content-based index structures will be introduced. The characteristics of different index structures will also be described by examining the methods for inserting data and splitting nodes. We will focus on R-Tree and its variant. First of all, R-Tree, which is a generalized B-Tree manipulating multi-dimensional data, will be described. Then, R$^+$-Tree and R*-Tree, which are variants of R-Tree, will also be discussed.

2.1 R-Tree

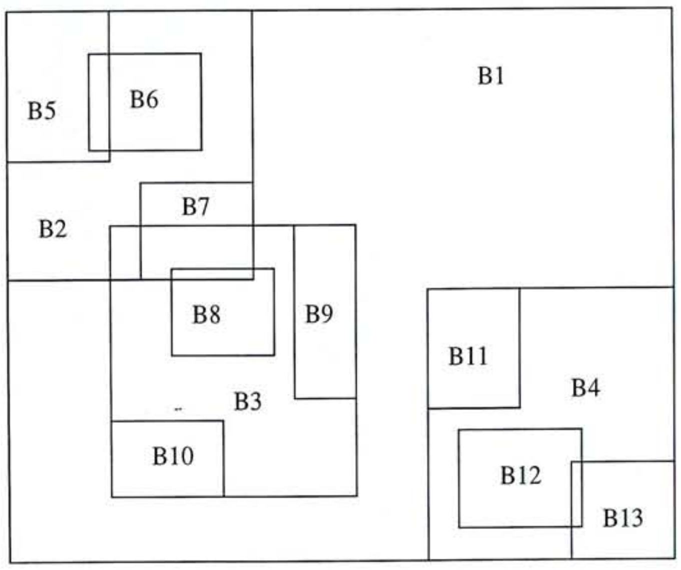


Figure 2.1: Example of R-Tree: Bounding box

A. Guttman in [4] proposed an multi-dimensional index structures called R-Tree. R-Tree is a generalization of B-Tree for multi-dimensional objects that are either points or regions. An R-Tree is a height-balanced tree similar to a B-Tree with index records in its leaf nodes containing pointers to data objects. We may assume that nodes correspond to disk pages if the index is disk-resident, and the structure is designed so that only a small number of nodes will be visited for each search query.

Leaf nodes in an R-Tree contain index object entries of the form $(MBR, \textit{tuple-pointer})$ where *tuple-pointer* refers to a tuple in the database and MBR is an n-dimensional rectangle which is the bounding box of the spatial object indexed:

$$MBR = (I_0, I_1, ..., I_{n-1})$$

Here n is the number of dimensions and I_i is a closed bounded interval $[a, b]$ describing the content of the object along dimension i. Non-leaf nodes contain

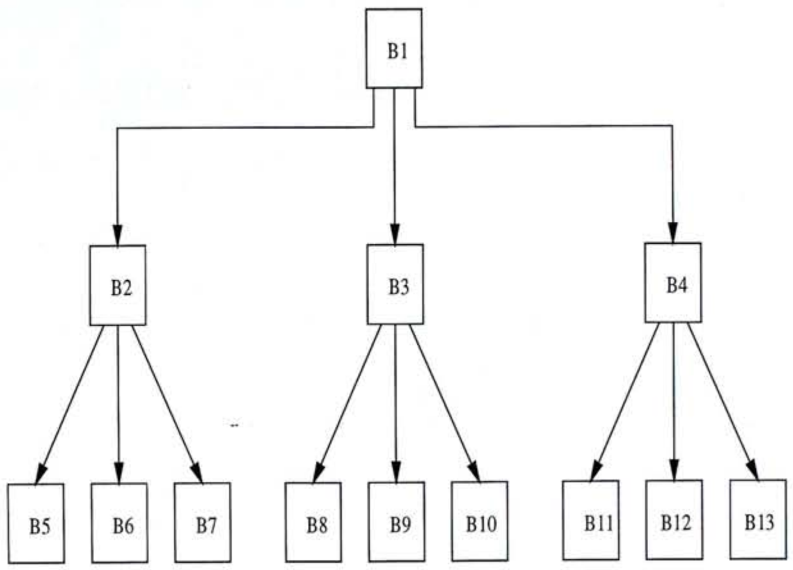


Figure 2.2: Example of R-Tree: Index structure

entries of the form

$$(MBR, child - pointer)$$

where $child - pointer$ is the address of a lower node in the R-Tree and MBR covers all bounding boxes in the lower node's entries. MBR is also referred as *Minimum Bounding Region* or *Minimum Bounding Box.*

Let M be the maximum number of entries that will fit in one node and let m be a parameter specifying the minimum number of entries in a node where

$$m \leq \frac{M}{2}$$

According to [4], an R-Tree must satisfy the following properties:

1. Every leaf node contains between m and M index entries unless it is the root.

2. For each index record *(MBR, tuple-pointer)* in a leaf node, *MBR* is the smallest bounding box that spatially contains the n-dimensional data object represented by the indicated tuple.

3. Every non-leaf node has between m and M child nodes unless it is the root.

4. For each entry *(MBR, child-pointer)* in a non-leaf node, *MBR* is the smallest bounding box that spatially contains the bounding boxes in the child node.

5. The root node has at least two child nodes unless it is a leaf.

6. All leaves appear on the same level.

Inserting objects for new data tuples is similar to insertion in a B-Tree. Started from the root of the tree, a child node which causes least area enlargement will be selected, and the process repeated until it is the leaf. Then, the new objects are added to the leaves. In case overflow of node occurs, nodes will be splitted and the splits propagate up the tree. The split should be done in a way that both new nodes will need to be examined on subsequent searches. Since the decision on whether to visit a node depends on whether its bounding box overlaps with the search area, the total area of the two bounding boxes after a split should be minimized. Several split algorithms have been discussed and the linear-cost split algorithm is suggested in [4] because it can give satisfactory splits while it does not involve too much computation.

The searching algorithm descends the tree from the root which is similar to a B-Tree. Details about searching will be described in Chapter 3.

2.2 R$^+$-Tree

If R-Tree is built using its insertion algorithms, the structure may provide a lot of overlapping regions and dead-space in the nodes that result in bad performance on data retrieval. In fact, the concepts of coverage and overlap are important, but they are not much concerned in R-Tree. Coverage of a level is defined as

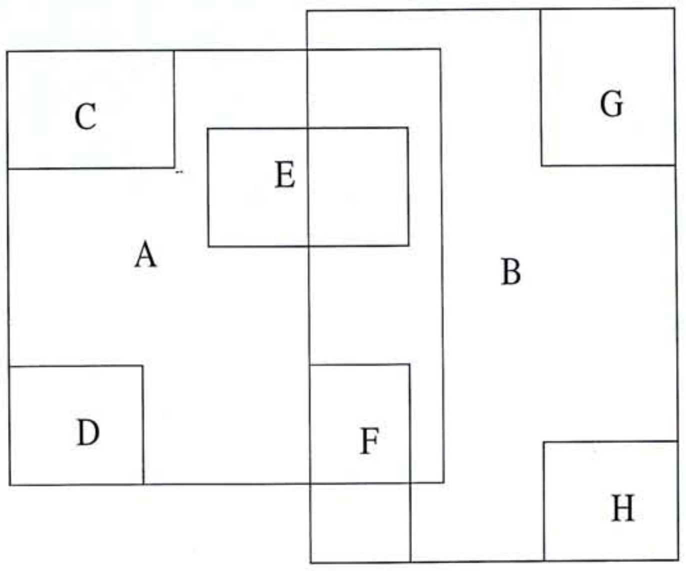


Figure 2.3: Bounding boxes before splitting

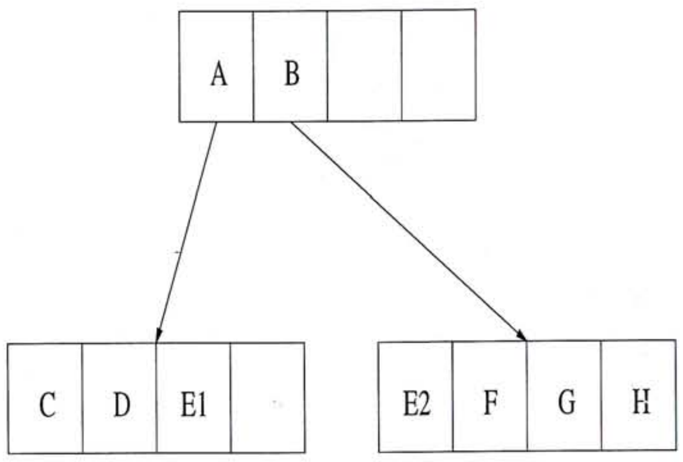


Figure 2.4: Example of R$^+$-Tree

the total area of all the bounding boxes associated with the nodes. Overlap of a level is defined as the total area contained within two or more nodes. Efficient R-Tree searching algorithms require that both overlap and coverage should be minimized. Minimal overlap is even more critical than minimal coverage. It has been shown that zero overlap and coverage is only achievable for data points that using a packing technique for R-Trees. The performance of searching is improved. Overlap-free splitting does not always exist for extended data objects.

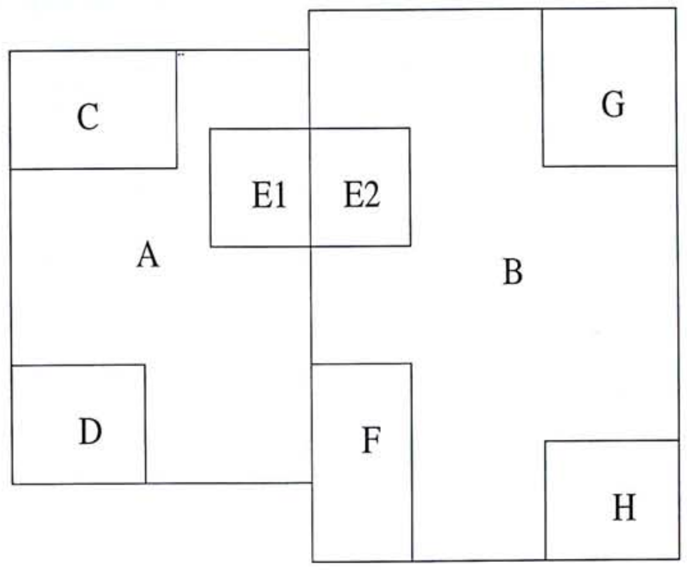


Figure 2.5: Bounding boxes of R$^+$-Tree

However, if partition is allowed to split bounding boxes, then overlap-free among intermediate nodes can be achieved. Whenever a bounding box at a lower level overlaps with another bounding box, it will be decomposed into a collection of non-overlapping boxes whose union is equal to the original bounding box.

In [5], T. Sellis *et al* proposed a variation of R-Tree which is called R$^+$-Tree. R$^+$-Tree avoids overlapping bounding boxes in intermediate nodes of the tree. The data structure of R$^+$-Tree is similar to R-Tree: Leaf nodes in an R$^+$-Tree contain index object entries of the form $(MBR, tuple-pointer)$. Non-leaf nodes contain entries of the form $(MBR, child-pointer)$. Besides, according to [5], R$^+$-Tree has following properties:

1. For each entry $(MBR, child{-}pointer)$ in an intermediate node, the subtree rooted at the node is pointed to by $child - pointer$ and it contains a bounding box R if and only if R is covered by MBR.

2. For any two entries $(MBR_1, child{-}pointer_1)$ and $(MBR_2, child{-}pointer_2)$ of an intermediate node, the overlap between MBR_1 and MBR_2 is zero.

3. The root has at least two children unless it is a leaf.

4. All leaves are at the same level.

There are differences between inserting an object to R$^+$-Tree and R-Tree. When an object is being inserted to R$^+$-Tree, it may be added to more than one leaf node. It is because it may be broken into several sub-regions. The splitting algorithm is more complicated than that of R-Tree because it avoids overlapping. If splitting is necessary, first of all, an splitting axis with smallest cost will be chosen. Next, it divides the space and distributes the child nodes. Then, it recursively packs the entries of each level of the tree so that overlap-free nodes are obtained. The split will be propagated upwards while the pack will be propagated downwards.

An example of R$^+$-Tree is shown in Figure 2.3. There are two overlapping bounding boxes A and B. Bounding box A encloses four bounding boxes: C, D, E and F. Bounding box B encloses four bounding boxes: E, F, G and H. Both bounding boxes A and B enclose bounding boxes E and F. In R-Tree, the overlapping region exists. It may happen that both E and F belong to same parent, say A, but the other bounding box B still encloses E and F. On the other hand, in R$^+$-Tree, overlapping is not allowed. E will be splitted into two nodes E_1 and E_2 such that E_1 is enclosed by A only and E_2 is enclosed by B only. Therefore, A and B can be adjusted to avoid overlapping region. Figure 2.4 shows the R$^+$-Tree and Figure 2.5 shows the bounding boxes in R$^+$-Tree of the example.

2.3 R*-Tree

R-Tree emphasizes on minimizing the area of each bounding box of nodes. N. Beckmann *et al* argues in [7] that it is neither the best nor the only optimization criteria. In R-tree, bounding boxes are built up from subsets of between m and M bounding boxes such that data retrieval operations are supported efficiently. However, the parameters used are not enough to provide efficient retrieval operations. The known parameters of good retrieval performance affect each other. N. Beckmann *et al* in [7] states four parameters for retrieval performance. They are area covered by a bounding box, overlap between bounding boxes, margin of a bounding box, and storage utilization. In order to achieve good performance, all four parameters should be optimized. However, it is impossible to optimize one of them without influencing other parameters which may cause a deterioration of the overall performance. In practice, only some of them can be optimized.

During the splitting phase of R-tree, all children of the splitting node will be distributed into two splitted nodes. When the distribution process begins, first of all, two entries, which are called seeds, among those children will be selected so that the pair would have the largest area if they are put in the same node. The pair are the first entries of those splitted nodes. Then remaining children of the splitting node will be distributed into these nodes by the least-enlargement-of-area strategy. No matter whether a quadratic or linear seed-picking algorithm which is recommended by Guttman in [4] is used, the splitting strategy is simple to implement. However, if small seeds are picked, several problems may occur. First of all, if in $d-1$ of the d dimensions a distant bounding box has nearly the same coordinates as one of the seeds, it will be distributed first. The area and the area enlargement of the created bounding box will be very small, but the actual distance is very large. In addition, the algorithm tends to favor the bounding box which is created from the first assignment of a bounding box to one seed. Moreover, if one group has reached the maximum number of entries $M-m+1$, all remaining entries are assigned to the other group without considering geometric properties.

R*-tree is a variant of R-tree. It is based on the reduction of the area, margin and overlap of the bounding boxes. In addition, it prevents splitting and its structure is reorganized dynamically. Thus, the storage utilization is higher than R-tree.

When choosing the appropriate subtree to insert data, one has to determine the minimum overlap cost, minimum margin cost and minimum area cost. Similarly, the minimum overlap cost is also concerned when splitting a node. When splitting occurs, it will have to choose a split axis, and choose a split index, before distributing entries into two groups. Choosing split axis is to determine the axis which is perpendicular to which the split is performed. The axis will be chosen if it has the greatest normalized separation of two most distant bounding boxes of the current node. Choosing split index is to choose the distribution with the minimum overlap-value along the chosen split axis. The choosing split axis and split index are very important to the performance of R*-tree because it is related to the presence of overlapping node in the tree. The split algorithm does not guarantee that an overlap-free split will be provided. However, R^+-tree cannot perform better than R*-tree because R^+-tree tries to provide overlap-free split which may make those nodes become less quadratic and hence their margin will be increased. In [7], margin of bounding box is one of four parameters that will affect the performance of the tree and it proves why R*-tree can perform better than R^+-tree.

Both R-tree and R*-tree are nondeterministic in allocation of the entries onto the nodes. R-tree forces entries to be reinserted during the deletion routine. R*-tree, on the other hand, forces entries to be reinserted during the insertion routine. There are some advantages of reinsertion algorithm in R*-tree. First of all, forced reinsert changes entries between neighboring nodes so that it decreases the overlap and improves the storage utilization. Moreover, the shape of the bounding boxes will be more quadratic.

N. Beckmann em et al [7] have done some experiments comparing the performance of Linear cost R-tree, Quadratic cost R-tree, Greene's R-tree, and R*-Tree. The results show that R*-Tree is the most outstanding algorithm. R*-Tree has also been proved to be the most efficient R-Tree variant on data retrieval. Many real applications, for example the QPIC project in [14], [15], [16] and [17], use R*-Tree to index their multi-dimensional multi-media data.

Chapter 3

Searching in Both R-Tree and R*-Tree

The main purpose of constructing a R-Tree is to index data so that it can provide efficient data retrieval. Besides the construction of R-Tree, its searching algorithms will also be described in this Chapter. Exact search and nearest neighbor search, which are the most commonly used searching method, will be introduced. Their algorithms and other related issues such as metrics and pruning heuristics used on nearest neighbor search will also be briefly explained.

3.1 Exact Search

Before the exact search algorithm is described, the definition of exact search should be clarified first: when an exact search query is to be performed, the index tree will be searched to see whether the given query object does exist in the tree or not. If it does, then the object is returned else a failure message is returned. That means, a query object will be specified and the searching operation is to check its existence in the tree.

The algorithm of exacting search is given in Algorithm 3.1.

Algorithm 3.1 Exact search algorithm for R-Tree

```
Procedure Exact_Search
Intput : Q
        /* Exact search query */
Output : Result
         /* Boolean value to show the query is found or not */
Begin
    If current node P is at leaf level
      If P is equal to the query Q
      Return P
    Else
        For i := 1 to number of children of P
            Set $P_i$ to be the $i^{th}$ children of P
            If the bounding box of $P_i$ encloses the query Q
            Then
                Call Exact_Search
                If the result is not equal to FALSE
                  Return the result
    Return FALSE
End
```

The exact search algorithm is based on the inside-outside test. From the property of R-Tree, all bounding boxes of child nodes will be enclosed by the bounding box of the parent node. Equivalently, if the bounding box of a node A is not enclosed by the bounding box of another node B, then node B will not be an ancestor of node A. The exact search algorithm makes use of this property to find the object: If the bounding of a node encloses the query object, then the node may contain the query object. Therefore, the node will be accessed and the process will be repeated until either the leaf level is reached or all bounding boxes of child nodes do not enclose the query. The process will be finished when the query object is found, or all nodes whose bounding boxes enclose the query object are searched but the query object is not found.

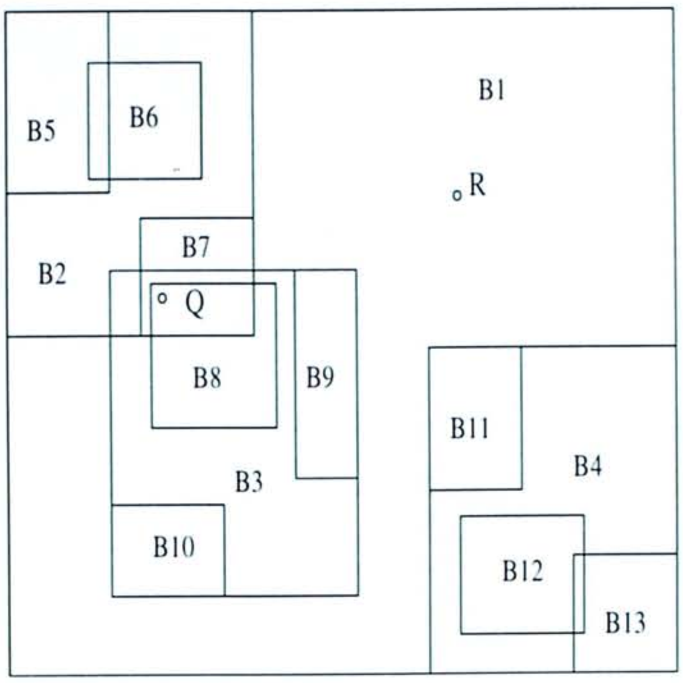


Figure 3.1: Example of exact search

An example on exact search is shown in Figure 3.1. A query Q is specified and we are going to search for the existence of Q in the Tree. Figure 3.2 shows the R-Tree structure of the example. The searching will be started at the root of tree, B_1. Since the bounding box of B_1 enclose the query, the searching will be continued and the child nodes of B_1 will be examined. The bounding box of the first child node of B_1, which is B_2, encloses the query. B_2 is then searched. It is found that all child nodes of B_2 do not contain the query even though there exists a child node of B_2, B_7, whose bounding box encloses the query. After examining the subtree of B_2 and the query is not found, the searching will be shifted to the subtree of B_3 whose bounding box encloses the query too. Suppose its child node B_8 contains the query, the searching will be terminated after the query is found and the result will be given to the user.

On the contrary, another query R is to be searched in the same example. The searching will also be started at the root of tree, B_1. Since the bounding box

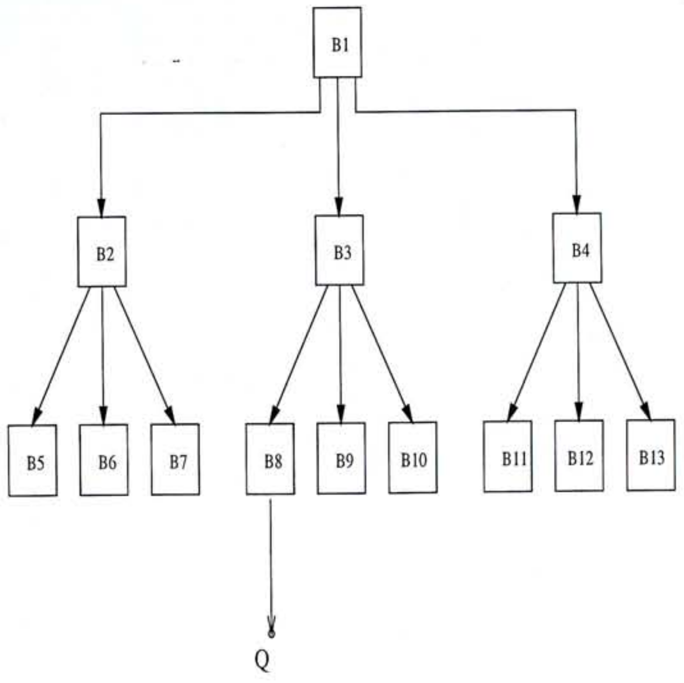


Figure 3.2: Exact search on R-Tree

of B_1 encloses the query, the searching will be continued and the child nodes of B_1 will be examined. It is found that all child nodes of B_1 do not contain the query, and therefore the searching will be terminated and a failure message will be returned.

Y.Theodoridis and T. Sellis in [24] proposed an equation to predict the performance of R-Tree in range search. Since an exact search query is equal to a range search query with a point-size query window, the performance of R-Tree in exact search can be derived as follows:

$$DiskAccess = \sum_{j=1}^{1+\lceil log_f \frac{N}{f} \rceil} \left\{ \frac{N}{f^j} \cdot \prod_{i=1}^{n} \left(D_j \cdot \frac{f^j}{N} \right)^{\frac{1}{n}} \right\} + 1$$

where the definition of symbols used in the above equation is explained in Table 3.1.

Symbols	Definition
n	number of dimensions
N	number of data
D_j	density of the dataset at level j
f	average node capacity
N_j	number of R-Tree nodes at level j

Table 3.1: Table of symbols and definition on equation of exact search

3.2 Nearest Neighbor Search

Nearest Neighbor search aims at searching for an object which is the nearest one among all data objects, to the query object. The meaning of nearest is usually the shortest Euclidean distance. The searching process is not as trivial as exact search. During the exact searching, the required object is known, and the main task is checking whether the object does exist in the index tree or not. On the other hand, in nearest neighbor search, the required object may not be the same as the query. Consequently, the nearest neighbor algorithm is more complicated than that of exact search, and it generally needs more disk access and time for processing.

3.2.1 Definition of Searching Metrics

Instead of using the trivial sequential search to find the nearest neighbor, Roussopoulos, Kelley and Vincent in [6] suggested an efficient nearest neighbor search algorithm on R-Tree. In this algorithm, an efficient pruning heuristics are used to discard impossible candidates from the Active Branch List, which stores all entries to be accessed, so that less nodes will be accessed and the correct result can be guaranteed at the same time. Before the nearest neighbor search algorithm is described, two metrics are introduced first. They are $MINDIST$, *minimum distance*, and $MINMAXDIST$, *minimum of maximum possible distances.*

Symbols	Definition
n	number of dimension
s_i	lower bound of the node on i^{th} dimension
t_i	upper bound of the node on i^{th} dimension
q_i	vector component of query point on i^{th} dimension

Table 3.2: Table of symbols and definition on nearest neighbor search

The first metric, $MINDIST_A$ which is the minimum distance from node A to the query $Q = \{q_1, q_2, ..., q_n\}$, is defined as follows:

Definition 1 *$MINDIST_A$, the minimum distance from bounding box of A to the query Q, is:*

$$MINDIST_A = \sum_{i=1}^{n} |q_i - r_i|^2$$

where

$$r_i = \begin{cases} s_i & \text{if } q_i < s_i \\ t_i & \text{if } q_i > t_i \\ q_i & \text{otherwise} \end{cases}$$

and s_i, t_i are defined in Table 3.2.

□

$MINDIST_A$ is the square of the minimum Euclidean distance from the node A to the query. Furthermore, it also serves as a lower bound of distance from the nearest neighbor within bounding box of node A to the query. That means, if an object P which is the nearest to the query among all objects in node A, then

$$MINDIST_A \leq DIST_P$$

must be true where $DIST_P$ is the distance from P to the query.

The second metric, $MINMAXDIST_A$, which is the minimum of maximum possible distances, is defined as follows:

Definition 2 *$MINMAXDIST_A$, the minimum of maximum possible distances of the bounding box of A from the query $Q = \{q_1, q_2, ..., q_n\}$, is:*

$$MINMAXDIST_A = \min_{1 \le k \le n} \left(|q_k - rm_k|^2 + \sum_{\substack{i \ne k \\ 1 \le i \le n}} |q_i - rM_i|^2\right)$$

where

$$rm_k = \begin{cases} s_k & \text{if } q_k \le \frac{(s_k+t_k)}{2} \\ t_k & \text{otherwise} \end{cases}$$

and

$$rM_i = \begin{cases} s_i & \text{if } q_i \ge \frac{(s_i+t_i)}{2} \\ t_i & \text{otherwise} \end{cases}$$

and s_k, t_k are defined in Table 3.2.

□

$MINMAXDIST_A$ serves as an upper bound of distance of the nearest neighbor in bounding box of node A to the query, and it was proved to be true by using the Minimum Bounding Region Face Property stated in [6]. Therefore, if P is an object nearest to the query among all objects in A, then

$$DIST_P \le MINMAXDIST_A$$

must be true too.

By the definitions of $MINDIST_A$ and $MINMAXDIST_A$, they serve as the bounds to the distance of nearest object in the bounding box of node A. It is particularly important to the nearest neighbor algorithm in [6] that can make use of the approximation to provide powerful pruning heuristics so that an efficient nearest neighbor search algorithm can be obtained.

3.2.2 Pruning Heuristics

In order to reduce the number of disk access during the nearest neighbor search, pruning heuristics have been used. Roussopoulos, Kelley and Vincent in [6] used the following three heuristics to discard nodes which are not or do not contain the nearest neighbor.

Let Q be a nearest neighbor query. Let $MINDIST_A$ and $MINDIST_C$ be the minimum distances from the query to minimum bounding boxes of nodes A and C respectively. Let $MINMAXDIST_A$ and $MINMAXDIST_C$ be the $MINMAXDIST$s from the query to minimum bounding boxes of nodes A and C respectively. Let P and B be objects at leaf level $DIST_P$ be the actual distance from P to Q and $DIST_B$ be the actual distance from B to Q.

Heuristic 1 *If $MINDIST_A$ is larger than $MINMAXDIST_B$, then node A will be discarded.*

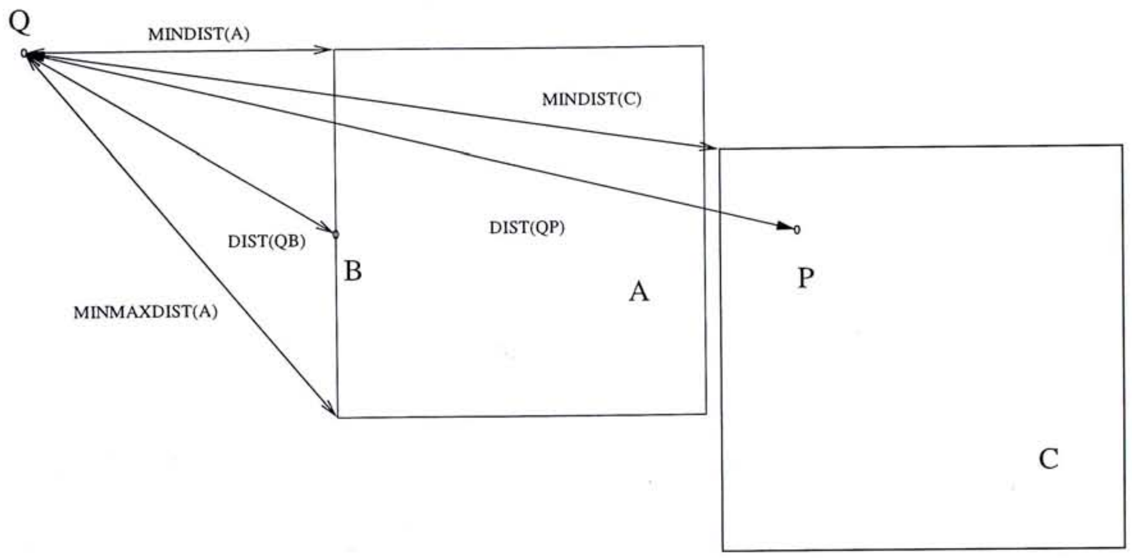


Figure 3.3: Discard node C by applying Heuristic 1

An example of using Heuristic 1 is shown in Figure 3.3. In this example, the $MINDIST$ of node C is larger than the $MINMAXDIST$ of node A. Since $MINDIST_A$ and $MINMAXDIST_A$ serve as the lower and upper bound of distance of the nearest neighbor within the bounding box of node A to the given query, Heuristic 1 can be used to discard node C from the Active Branch List because its nearest neighbor to the query, say P, must be farther than object B which is the nearest neighbor in node A to the query.

Heuristic 2 *If $DIST_P$ is larger than $MINMAXDIST_B$, then the object P will be discarded.*

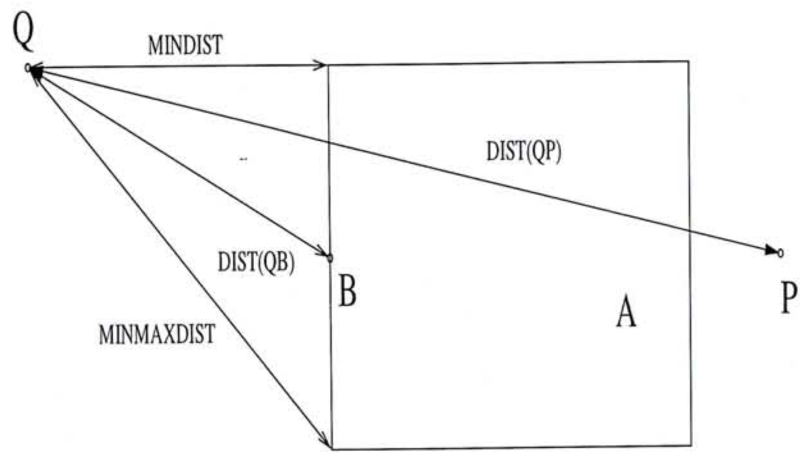


Figure 3.4: Discard object P by applying Heuristic 2

Figure 3.4 shows an example of using Heuristic 2. Similar to the example used in Figure 3.3, Heuristic 2 can be applied to discard object P from the Active Branch List. It is because its distance to the query is larger than the upper bound of nearest neighbor of node A, and consequently, it must be farther than object B which is the nearest neighbor of node A to the query.

Heuristic 3 *If $MINDIST_A$ is larger than $DIST_P$, then node A will be discarded.*

An example of using Heuristic 3 is shown in Figure 3.5. In this example, the $MINDIST_A$ is larger than the distance from the query to an object P, which may be the nearest neighbor of the given query. Using Heuristic 3, node A will be discarded from the Active Branch List because it is the nearest neighbor to the query, say B, must be farther than object P which implies all child nodes of A must not be the nearest neighbor to the query.

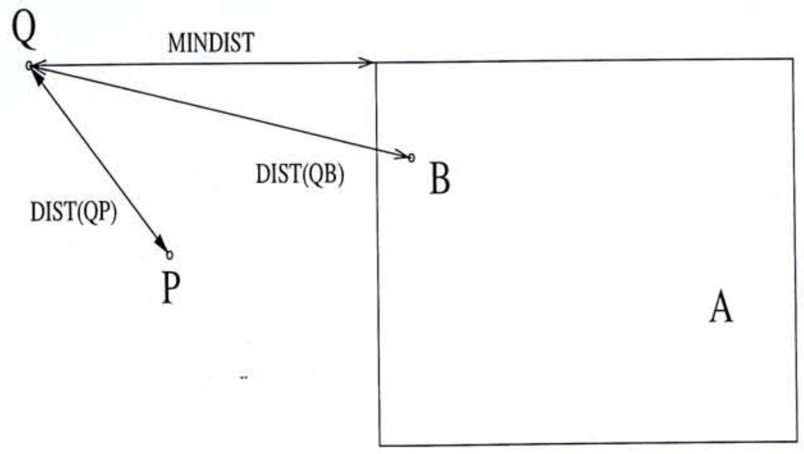


Figure 3.5: Discard node A by applying Heuristic 3

3.2.3 Nearest Neighbor Search Algorithm

After the metrics and pruning heuristics have been introduced, we present the nearest neighbor search algorithm in [6]. The nearest neighbor search algorithm is given in Algorithm 3.2.

In the algorithm, the current node will first be checked to see whether it is at the leaf level or not. If it is a leaf, then its distance to the query will be calculated, and if the distance is less than the distance from the temporary nearest neighbor, then we set the temporary nearest neighbor to be current node and update the nearest distance.

On the other hand, if the current node is not at the leaf level, then the Active Branch List for further search will be generated. The Active Branch List is a list which contains all child nodes of current node that will be accessed in order to get the nearest neighbor. The Active Branch List is sorted by ascending order of $MINDIST$. Next, pruning will be performed by applying Heuristics 1 and 2. Then, it iterates through the Active Branch List and recursively access the child nodes by calling NN_Search. After NN_Search has been called, pruning will be performed by applying Heuristic 3. The recursive call and the second pruning

Algorithm 3.2 Nearest neighbor search algorithm for R-Tree

```
Procedure NN_Search
Input : NODE
        /* node to be visited */
        $NN\_DIST_{temp}$
        /* distance from temporary nearest neighbor to the query */
Begin
    If current node P is at leaf level
    Then
        For i := 1 to no. of children of current node
            If $DIST_P < NN\_DIST_{temp}$
                Set current node to be nearest neighbor
                Update $NN\_DIST_{temp}$
    Else
        Generate Active Branch List of current node

        Calculate $MINDIST$ and $MINMAXDIST$

        Sort the Active Branch List by ascending ordering of $MINDIST$

        Apply Heuristic 1 and 2 to prune objects

        For i := 1 to no. of entries in the Active Branch List
            Call $NN\_Search$
            Apply Heuristic 3 to prune objects
End
```

will be repeated until no entry in the Active Branch List remains unvisited.

3.2.4 Generalization to N-Nearest Neighbor Search

Roussopoulos *et al* proposed an nearest neighbor search algorithm for R-Tree and its variant in [6]. They also described how the N-nearest neighbor search algorithm can be derived from their nearest neighbor search algorithm. The followings must be considered:

- A sorted buffer of at most N temporary nearest neighbors is needed.

Algorithm 3.3 N-Nearest neighbor search algorithm for R-Tree

```
Procedure N-NN_Search
Input : NODE
        /* node to be visited */
        $NN\_DIST_N$
        /* distance from temporary N-nearest neighbor to the query */
Begin
    If current node P is at leaf level
    Then
        For i := 1 to no. of children of current node
            If $DIST_P < NN\_DIST_N$
                Call $Insert\_NN(P)$
                Update $NN\_DIST_N$
    Else
        Generate Active Branch List of current node

        Calculate $MINDIST$ and $MINMAXDIST$

        Sort the Active Branch List by ascending ordering of $MINDIST$

        Apply Heuristic 1 and 2 to prune objects

        For i := 1 to no. of entries in the Active Branch List
            Call $N\text{-}NN\_Search$
            Apply Heuristic 4 to prune objects
End
```

- The minimum bounding boxes pruning is done according to the distance of the furthest nearest neighbor in this buffer.

Based on the nearest neighbor search algorithm in [6], we can extend the algorithm to find the N-nearest neighbor of a given query.

We define that the first nearest neighbor is the closest object to the query while the N^{th} nearest neighbor is the farthest object among all entries in the nearest neighbor list. Let the distance from the first nearest neighbor to the query denoted by NN_DIST_1, the distance from the second nearest neighbor to the query denoted by NN_DIST_2, and so on. In the algorithm, we implement a nearest neighbor list which acts as a buffer and it stores the temporary N-nearest neighbors of the query. The nearest neighbor list is an ordered list. When an object has been checked that it should be a member of the temporary N-nearest neighbors, a procedure $Insert\_NN(P)$ will be invoked to insert the object into an appropriate position in the list.

In Roussopoulos's algorithm, NN_DIST_{temp} stores the distance from the temporary nearest neighbor to the query. In N-nearest neighbor search algorithm, however, NN_DIST_{temp} is not adequate because it does not give any information about the temporary Nth nearest neighbor which should be used for pruning unnecessary nodes. A new pruning heuristic should be designed for N-nearest neighbor search algorithm. The new heuristic is based on the nearest neighbor algorithm except that NN_DIST_{temp} is replaced by NN_DIST_N which is the temporary N^{th} nearest neighbor of the query. The following heuristic is the new pruning heuristic.

Heuristic 4 *If $MINDIST_A$ is larger than NN_DIST_N, then node A will be discarded.*

Theorem 1 *Heuristic 4 is true.*

Proof: By definition,

$$NN\_DIST_1 \leq NN\_DIST_2 \leq ... \leq NN\_DIST_N$$

is true. Therefore, if $NN\_DIST_N < MINDIST_A$ where $MINDIST_A$ is the minimum distance from bounding box of node A to the query, all objects which are enclosed by the bounding box of node A must be farther than the N^{th} nearest neighbor whose distance to the query is NN_DIST_N. Hence, Heuristic 4 is proved to be correct. □

In the algorithm, the current node will first be checked to see whether it is at the leaf level or not. If it is a leaf, then its distance to the query will be calculated, and if the distance is less than the distance from the temporary nearest neighbor, then we set the temporary nearest neighbor to be current node and update the nearest distance. On the other hand, if the current node is not at the leaf level, then the Active Branch List for further search will be generated. The Active Branch List is a list which contains all child nodes of current node that will be accessed in order to get the nearest neighbor. $MINDIST$ and $MINMAXDIST$ are calculated for each entry in the Active Branch List, and the Active Branch List is sorted by ascending order of $MINDIST$. Heuristics 1 and 2 are applied to prune objects. Then, it iterates through the Active Branch List and recursively access child nodes by calling NN_Search. After NN_Search has been called, pruning will be performed by applying Heuristic 4. The recursive call and the second pruning will be repeated until no entry in the Active Branch List remains unvisited.

Chapter 4

An Improved Nearest Neighbor Search Algorithm for R-Tree

4.1 Introduction

The task of nearest neighbor search is finding the nearest neighbor from a set of data for a nearest neighbor search query. Nearest neighbor search is not as easy as exact search. In general, it needs more node access than exact search because it may continue the search process even though the real nearest neighbor is found until the identification of the nearest neighbor is proved. It is obvious that a good estimation of nearest neighbor and a set of efficient pruning heuristics are very important to the performance of the search.

In Chapter 3, we have briefly described two metrics, *MINDIST* and *MINMAXDIST* and three pruning heuristics which have been used on nearest neighbor search in [6]. Those heuristics can reduce the number of node access on an R-Tree and its variants when it is compared to linear search on the dataset. However, extra CPU time overhead will be introduced by the process of calculating the two metrics. Algorithm 4.1 shows an algorithm of calculating the

Algorithm 4.1 Calculation of $MINMAXDIST$

```
Procedure CALCULATE_MINMAXDIST
Input : $(\{s_1, t_1\}, \{s_2, t_2\}, ..., \{s_n, t_n\})$
        /* The bounding box of the node */
Output : $MINMAXDIST$
        /* The $MINMAXDIST$ of the node */
Begin
    Initialize $MINMAXDIST$

    Set Dim to be number of dimension

    For i := 1 to Dim

        Set $MINMAXDIST_{temp}$ to be zero

        For k := 1 to Dim
            If i = k
                Then If $q_k \leq \frac{(s_k+t_k)}{2}$
                        Then $rm_k := s_k$
                        Else $rm_k := t_k$

                    $MINMAXDIST_{temp} := MINMAXDIST_{temp} + |q_k - rm_k|^2$
            Else If $q_k \geq \frac{(s_k+t_k)}{2}$
                    Then $rM_k := s_k$
                    Else $rM_k := t_k$

                $MINMAXDIST_{temp} := MINMAXDIST_{temp} + |q_k - rM_k|^2$

        If $MINMAXDIST_{temp} < MINMAXDIST$
            Then $MINMAXDIST := MINMAXDIST_{temp}$
End
```

MINMAXDIST of a node. It is easy to observe that the calculation of *MINMAXDIST* is computationally expensive and the time is bounded by $O(n^2)$ where n is the number of dimension. There are two pruning heuristics which make uses of *MINMAXDIST*. The overhead is large especially when a large amount of high dimension data has to be dealt with, which is common to occur

Symbols	Definition
NN_{real}	nearest neighbor to the given query
NN_{temp}	nearest neighbor among searched objects
$MINDIST_A$	minimum distance from node A to the query
$MINMAXDIST_A$	minimum of maximum possible distance from node A to the query
NN_DIST_{real}	distance from the nearest neighbor to the query
NN_DIST_{temp}	distance from the temporary nearest neighbor
$DIST_P$	distance from the object P to the query

Table 4.1: Table of symbols and definition

in real multi-media applications.

It is found that the calculation of *MINMAXDIST* is expensive. Since Heuristics 1 and 2 make use of *MINMAXDIST*, they should be replaced by a new pruning heuristic which does not use *MINMAXDIST* if the new heuristic can be derived so that CPU overhead can be reduced and the node access overhead will not be increased. If such a case exists, the old heuristic should be replaced by the more efficient one.

In this Chapter, new pruning heuristic and a new nearest neighbor search algorithm based on the new heuristic will be proposed. The new heuristic will be shown to be at least as powerful as the original heuristics in terms of the number of disk access during the searching. Therefore, the number of disk access during the searching will not be increased. Experimental results will be given to show that the total CPU time used for nearest neighbor search will be reduced when the new heuristic is applied.

4.2 New Pruning Heuristics

Two new pruning heuristics for nearest neighbor search on R-Tree will be introduced. Before the heuristics are described, lemmas, which will be used for proving the correctness and efficiency of those pruning heuristics, are described first.

Lemma 1 *If P is the nearest neighbor among all objects in node A to the query Q, then* $MINDIST_A \leq DIST_P \leq MINMAXDIST_A$.

Proof: By definition, $MINDIST_A$ is the minimum distance from A to the query Q. From the minimal bounding region face property shown in [6], if P is an object nearest to the query among all objects in A, then $DIST_P \leq MINMAXDIST_A$. Therefore, $MINDIST_A$ and $MINMAXDIST_A$ serve as a lower bound and a upper bound to the distance from the nearest neighbor in node A to the query respectively. □

Lemma 2 *If there are two nodes A and B with the condition* $MINDIST_A \leq MINDIST_B$*, then* $MINMAXDIST_B \nless MINDIST_A$.

Proof: From Lemma 1, $MINDIST_B \leq MINMAXDIST_B$ must be true for all nodes B. Since the precondition $MINDIST_A \leq MINDIST_B$ is provided, the following inequality $MINDIST_A \leq MINDIST_B \leq MINMAXDIST_B$ can be derived. Hence $MINMAXDIST_B \nless MINDIST_A$. □

Lemma 3 *If A is an ancestor node of B in a R-tree, then* $MINDIST_A \leq MINDIST_B$.

Proof: This follows from the definition of $MINDIST$. □

After some lemmas have been introduced, the new pruning heuristics for nearest neighbor search is to be described in detail. We assume that in the search algorithm that makes use of these heuristics, the distance to the nearest object discovered so far is kept in a variable NN_DIST_{temp}, that is, the distance from the temporary nearest neighbor to the query.

Heuristic 5 *If $MINDIST_A$ is greater than NN_DIST_{temp}, then node A will be discarded.*

Theorem 2 *Heuristic 5 is correct.*

Proof: $NN\_DIST_{real} \leq DIST_P$ for all objects P in the dataset. For all objects R which are under the child nodes of node A, by Lemma 1, $MINDIST_A \leq DIST_R$ must be true. If $MINDIST_A$ is greater than NN_DIST_{temp}, the relation $NN\_DIST_{temp} < DIST_R$ must also be true. Therefore, R must not be the nearest neighbor and it implies that all child nodes of A must not contain the real nearest neighbor. Consequently, node A can be discarded from the Active Branch List. Hence, the heuristic is correct. □

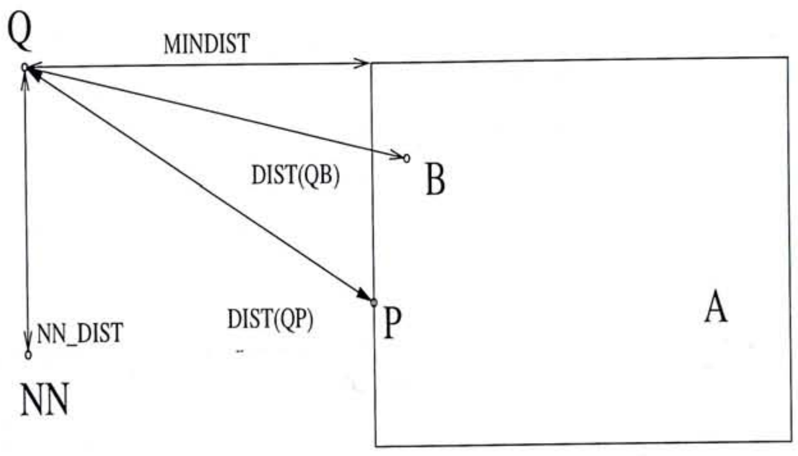


Figure 4.1: Example: discard node A by applying Heuristic 5

Figure 4.1 shows an example of applying Heuristic 5. In the figure, NN represents the temporary nearest neighbor to query among all searched objects. The condition

$$NN\_DIST_{temp} \leq MINDIST_A$$

is given. No matter whether NN is the real nearest neighbor to the query or not, the nearest neighbor in node A, say B, must have a greater distance to the query than NN. That means all child nodes of A cannot be the nearest neighbor. Therefore, node A can be removed from the Active Branch List by applying Heuristic 5.

4.3 An Improved Nearest Neighbor Search Algorithm

Algorithm 4.2 New nearest neighbor search algorithm for R-Tree

```
ALGORITHM B:
Procedure NN_Search
Input : NODE
        /* node to be visited */
        $NN\_DIST_{temp}$
        /* distance from temporary nearest neighbor to the query */
Begin
    If current node P is at leaf level
    Then
        If $DIST_P < NN\_DIST_{temp}$
            Set current node to be nearest neighbor
            Update $NN\_DIST_{temp}$
    Else
        Generate Active Branch List of NODE

        Calculate $MINDIST$

        Sort the Active Branch List by ascending ordering of $MINDIST$

        For i := 1 to no. of entries in the Active Branch List
            Apply Heuristic 5 to do pruning
            Call $NN\_Search$
End
```

After the new heuristic has been introduced and proved to be correct in the last section, a new nearest neighbor search algorithm is proposed that can make use of the new pruning heuristics. Let us denote the original algorithm in [6] by

Algorithm A which uses Heuristics 1, 2 and 3, and MINDIST ordering in the Active Branch List. The improved nearest neighbor search algorithm is given as **Algorithm B** (4.2).

Algorithm B is similar to the nearest neighbor search algorithm shown in Algorithm 3.2. The difference between the new and the original search algorithms is that Heuristics 2 and 3 have been replaced by Heuristic 5 in the new algorithm. Heuristic 1 have also been deleted from the algorithm.

In the new nearest neighbor search algorithm, first of all, the current node will be checked to see whether it is at leaf level or not. If it is a leaf node, then its distance to the query will be calculated. [1] If the distance is less than the distance from the temporary nearest neighbor to the query, then the temporary nearest neighbor is set to be the current node and the nearest distance is updated.

On the other hand, if the current node is not at leaf level, then the Active Branch List for further searching will be generated and all entries in the list will be sorted by ascending order of $MINDIST$. Next, it iterates through the Active Branch List and recursively access child nodes by calling NN_Search. Before NN_Search is called, pruning operations will be performed by applying Heuristic 5 so that unnecessary nodes will be pruned before accessed. The recursive procedure call and the pruning will be repeated until no entry in the Active Branch List remains unvisited.

In the nearest neighbor search algorithm in [6], there is a statement "It may

[1]Note that the algorithm in [6] assumes that a leaf node contains a number of objects. In our case we assume objects are N-dimensional points and the leaf node corresponds to a single object. However, it is straightforward to modify Algorithm B to assume leaf nodes contains objects that are not simple points, and the proofs of correctness and pruning ability will apply to the modified algorithm.

discard all entries in the Active Branch List." However, the statement is not true if Algorithm A is used. There exists at least one node, the first entry in the Active Branch List, to be accessed. The following example illustrate the problem. Assume a nearest neighbor search query is given. Current node A have a child node A' which is the first entry in the Active Branch List of node A and the following relations are true:

$$MINDIST_A \leq MINDIST'_A \leq MINMAXDIST_A \quad (1)$$

$$MINDIST_A < NNDIST_{temp} \quad (2)$$

$$NNDIST_{temp} < MINDIST'_A \quad (3)$$

That means, when node A is being accessed, (2) implies that A may contain the nearest neighbor. By (3), A' cannot contain the nearest neighbor. However, in Algorithm A, A' will also be accessed as it cannot be pruned by heuristics 1 and 2. Only heuristic 3 can prune A' from the Active Branch List, but, A' will be accessed before the heuristic is applied. Simply reversing the order of pruning and searching operation can solve the problem. Therefore, in Algorithm B, pruning is performed before further node access will be carried out.

4.4 Replacing Heuristics

In previous section, a new pruning heuristic for nearest neighbor search has been introduced. In order to replace the original three heuristics, it is necessary to show that the new heuristic is more efficient than the old one. In this section, we show that the number of node access will not be increased after the new heuristic is used instead of the old heuristics. That means, if a node is pruned by the old heuristics in the old algorithm, then it will also be pruned by the new heuristic in the new algorithm. We assume that node access corresponds to disk access. Once this is established, we can see that with the new algorithm, the computational cost will be dramatically decreased without increasing any disk access.

The first heuristic to be considered is Heuristic 2. Heuristic 2 says that if $DIST_P$ is greater than $MINMAXDIST_A$, then the object P will be discarded. Note that no node access is reduced in this case, since the discarded object P is already searched. (Effectively, if P is the nearest object discovered so far, then the NN_DIST_{temp} is updated to be $MINMAXDIST_A$.) Since we are interested here only in the reduction of node access, Heuristic 2 can be ignored.

The second heuristic we consider is Heuristic 3. Heuristic 3 says that if $DIST_P$ is smaller than $MINDIST_A$, then the node A will be discard.

Lemma 4 *If a node is pruned by Heuristic 3 using Algorithm A, it can be also be pruned by Algorithm B.*

Proof: Assume that during the execution of Algorithm A, there is a node A and an object P in the Active Branch List, and $MINDIST_A > DIST_P$, so that node A will be pruned by Heuristic 3. Next suppose Algorithm B is used, there are two possibilities.

Case 1: Node A is searched before P is either searched or pruned. Since P and A have a common root, and A is not the root, then an ancestor of P must be searched before A, let this ancestor be P'. By Lemma 3, the ancestor P' must have a MINDIST smaller than $DIST_P$, and also the nodes in the path in the tree from P' to P must all have MINDIST smaller than $DIST_P$. If P is pruned before being searched, then A would also be pruned since the pruning is via Heuristic 5, and A has a greater MINDIST than P's ancestors. If P is not pruned, since a basic depth-first traversal with MINDIST ordering is followed in the nearest neighbor search, it is not possible that A is searched before P is searched. Therefore, we conclude that this case cannot happen.

Case 2: The object P is searched before node A is either searched or pruned. Hence P has been considered as a possible candidate for the temporary nearest neighbor. Let NN_DIST_{temp} be the distance of the nearest neighbor discovered immediately before the search of node A. Since updates in the temporary nearest neighbor can only get closer to the query point, $NN\_DIST_{temp} \leq DIST_P$ must be true. Since $MINDIST_A > DIST_P$, has been given, $NN\_DIST_{temp} < MINDIST_A$ can be derived and the node A will be pruned by Heuristic 5.

Therefore, if a node can be pruned by Heuristic 3 using Algorithm A, then it can also be pruned by Heuristic 5 in Algorithm B. □

We have just shown that if a node is pruned by applying Heuristic 3 in Algorithm A, then it will be pruned by Algorithm B. Since Heuristic 2 in Algorithm A does not do effective node pruning, it remains to be shown that every node which is pruned by Heuristic 1 in Algorithm A will also be pruned by Algorithm B. In order to do so, we would make use of the following lemma.

Lemma 5 *If a node B is searched before a sibling node A using Algorithm B, and*

$$MINMAXDIST_B < MINDIST_A,$$

then the distance of the temporary nearest neighbor, NN_DIST_{temp}, just before A is either searched or pruned is less than or equal to $MINMAXDIST_B$.

Proof: Let α be the set of nodes that are searched after B and before the search or pruning of A. (We say that A is pruned when either it is pruned or an ancestor node containing B is pruned.) Let B_C be the object in B that is closest to the query point. There are two possible cases:

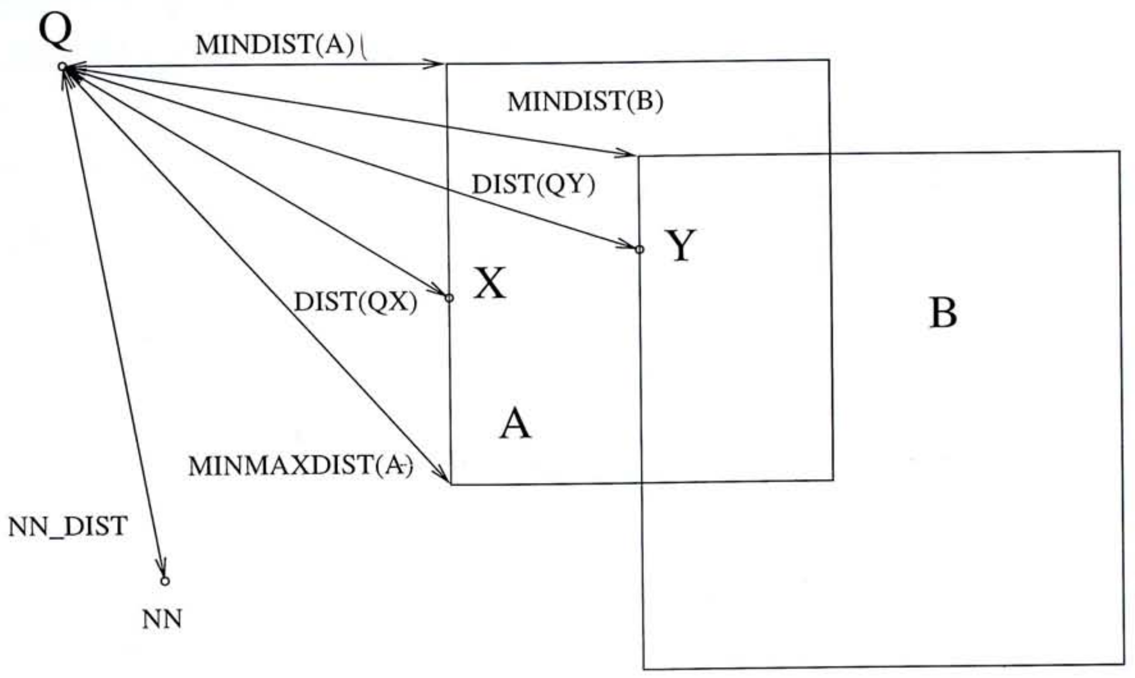


Figure 4.2: Example: pruning B from the Active Branch List

Case 1: B_C is in α. In this case, B_C has been considered as a candidate for the temporary nearest neighbor, then since we know that its distance is less than or equal to $MINMAXDIST_B$, hence $NN\_DIST_{temp} \leq MINMAXDIST_B$.

Case 2: B_C is not in α. Since $DIST_{B_C} \leq MINMAXDIST_B < MINDIST_A$, by Lemmas 1 and 3, all ancestor nodes of B_C have $MINDIST < MINDIST_A$. As B_C is not in α, an ancestor node of B, B', must have been in α and has been pruned by Heuristic 5. That is, node B' is discarded because $MINDIST_{B'} > NN\_DIST'_{temp}$. Hence $NN\_DIST_{temp} < MINDIST_{B'} \leq MINMAXDIST_A$. □

Lemma 6 *If a node is pruned by Heuristic 1 using Algorithm A, it will be pruned by Algorithm B.*

Proof: Heuristic 1 says that if $MINDIST_C$ is greater than $MINMAXDIST_D$ then node C is discarded. Without loss of generality, suppose there are two nodes A and B so that Node A is discarded by Heuristic 1 because of Node B in

Algorithm A. Hence A and B are sibling nodes (in the same active branch list) and $MINMAXDIST_B < MINDIST_A$. There are three cases to consider:

Case 1: $MINDIST_A \leq MINDIST_B$.
According to Lemma 1, we have inequalities $MINDIST_A \leq MINMAXDIST_A$ and $MINDIST_B \leq MINMAXDIST_B$. Since, $MINDIST_A \leq MINDIST_B$, by Lemma 2, we have $MINMAXDIST_B \nless MINDIST_A$. Therefore, it is impossible that $MINMAXDIST_B < MINDIST_A$ so that node A is pruned by Heuristic 1.

Case 2: $MINDIST_A > MINDIST_B$, and Node A is searched before node B in Algorithm B. This is not possible since the search is ordered by the values of MINDIST.

Case 3: $MINDIST_A > MINDIST_B$. Algorithm B is used and node B is searched before node A. Let NN_DIST_{temp} be the distance of the temporary nearest neighbor just before A is either searched or pruned. By Lemma 5,

$$NN\_DIST_{temp} \leq MINMAXDIST_A$$

Since the condition $MINMAXDIST_B < MINDIST_A$ is given, the relation

$$NN\_DIST_{temp} < MINDIST_A$$

can be derived from the above inequalities. Therefore, node A will be pruned by Heuristic 5.

The above show that all nodes pruned by Heuristic 1 in Algorithm A will be pruned by the new heuristic using Algorithm B. □

Theorem 3 *If node access corresponds to disk access, then Algorithm B requires no extra disk access compared to Algorithm A.*

Proof: Under our assumption, for a given R-tree, disk access is required if a node is searched for the first time. Hence the theorem follows directly from Lemmas 4 and 6. □

From Theorem 3, we find that Heuristics 1, 2, and 3 can be replaced by the new heuristic without deteriorating the performance of nearest neighbor search in terms of disk access. Consequently, *MINMAXDIST* is not a must to be calculated as the new heuristic does not make use of it. Therefore, a large amount of computation cost can be saved.

4.5 N-Nearest Neighbor Search

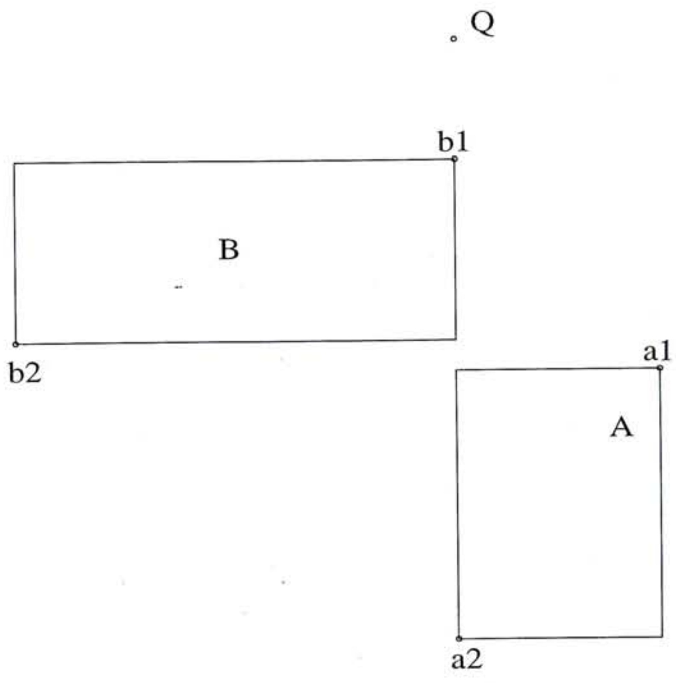


Figure 4.3: Node A cannot be discard

In Chapter 3, we have presented an N-nearest neighbor search algorithm that is a strict extension from the nearest neighbor search algorithm in [6]. In this section, We are going to present our proposed N-nearest neighbor algorithm for

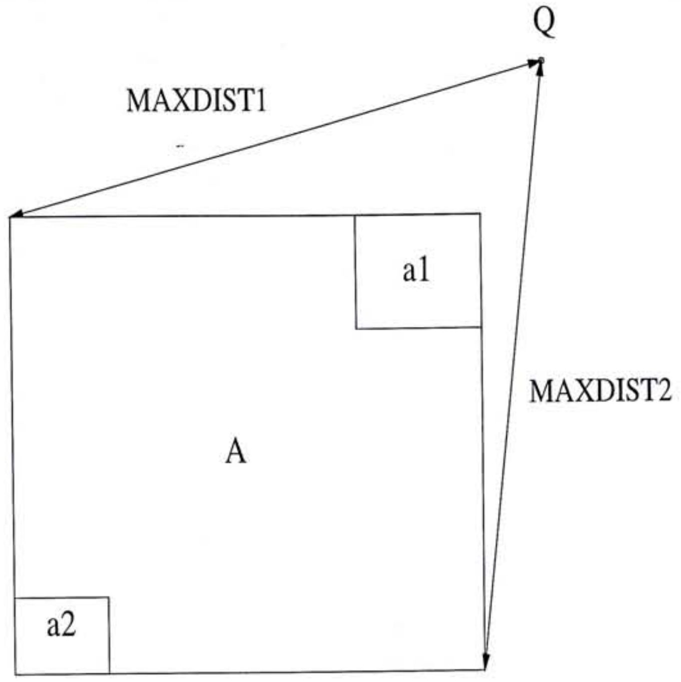


Figure 4.4: N-nearest neighbor search

R-Tree which is based on our improved nearest neighbor search algorithm.

In nearest neighbor search, $MINMAXDIST_A$ serves as an approximation to the nearest neighbor of a query such that it is the upper bound of the distance between the query and the nearest object in node A. On the other hand, $MINMAXDIST_A$ does not serve as an approximation to the N-nearest neighbor of the query. One may argue that we use $N\text{-}MINMAXDIST_A$, the Nth minimum of maximum possible distance from the node A to the query, to play a similar role to $MINMAXDIST_A$ in nearest neighbor search. However, there are two problems if $k\text{-}MINMAXDIST_A$ is used. First of all, the number of dimension must be larger or equal to N. Secondly, $N\text{-}MINMAXDIST_A$ may not be the upper bound of the distance between the N-nearest neighbor and the query. An example is given in Figure 4.4. In this example, there

Algorithm 4.3 New N-Nearest neighbor search algorithm for R-Tree

```
Procedure N-NN_Search
Input : NODE
        /* node to be visited */
        $NN\_DIST_N$
        /* distance from temporary N-nearest neighbor to the query */
Begin
    If current node P is at leaf level
    Then
        For i := 1 to no. of children of current node
            If $DIST_P < NN\_DIST_N$
                Call $Insert\_NN(P)$
                Update $NN\_DIST_N$
    Else
        Generate Active Branch List of current node

        Calculate $MINDIST$

        Sort the Active Branch List by ascending ordering of $MINDIST$

        For i := 1 to no. of entries in the Active Branch List
            Apply Heuristic 4 to prune objects
            Call $N\text{-}NN\_Search$
End
```

is a 2-dimensional bounding box A which have two children a_1 and a_2. A 2-nearest neighbor query Q has been given. $MAXDIST_1$ is the maximum possible distance from the boundary of dimension 1 of node A to the query while $MAXDIST_2$ is the maximum possible distance from the boundary of dimension 2 of node A to the query. Note that $MAXDIST_1$ and $MAXDIST_2$ are smaller than $MINDIST_{a_2}$. Therefore, the nearest object in node A will not be accessed even though it may be one of the N-nearest neighbors. The example shows that $N\text{-}MINMAXDIST_A$ cannot be used in the N-nearest neighbor search algorithm. Consequently, we only use Heuristic 4 to do the pruning operation.

In Chapter 3, we have proved that Heuristic 4 is true. In our N-nearest neighbor search algorithm, we will also use this pruning heuristic. On the contrary, Heuristics 1 and 2 will not be used because $MINMAXDIST$ is useless in N-nearest neighbor search. Heuristic 1 states that if $MINMAXDIST_B \leq MINDIST_A$, then node A will be discard. However, in N-nearest neighbor search, the pruning heuristic is not necessarily true. Assume 2-nearest neighbor search is being performed and a query Q has been specified. Figure 4.3 shows the example. Node A has two child nodes a_1 and a_2; Node B has two child nodes b_1 and b_2. The 2-nearest neighbor should be a_1 and b_1. Therefore, node A cannot be pruned even though $MINMAXDIST_B \leq MINDIST_A$ and hence Heuristic 1 cannot be used. Similar argument can be applied to Heuristic 2.

In the algorithm, the current node will first be checked to see whether it is at the leaf level or not. If it is a leaf, then its distance to the query will be calculated, and if the distance is less than the distance from the N^{th} nearest neighbor, NN_DIST_N, $Insert\_NN$ will be invoked to insert the object in the nearest neighbor list and then we update NN_DIST_N. On the other hand, if the current node is not at the leaf level, then the Active Branch List for further search will be generated. The Active Branch List is a list which contains all child nodes of current node that will be accessed in order to get the nearest neighbor. The Active Branch List is sorted by ascending order of $MINDIST$. Next, it iterates through the Active Branch List and recursively access child nodes by calling NN_Search. After NN_Search has been called, pruning will be performed by applying Heuristic 4. The recursive call and the second pruning will be repeated until no entry in the Active Branch List remains unvisited.

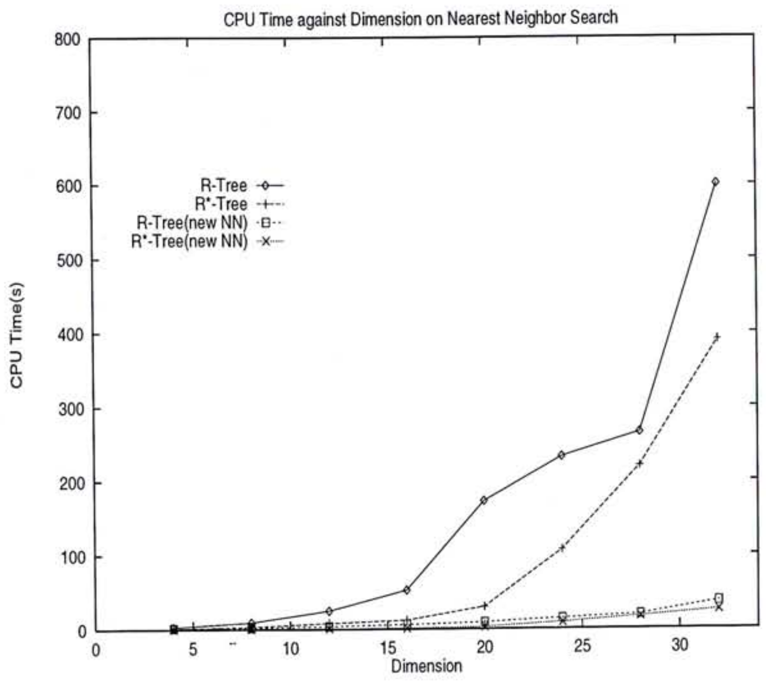


Figure 4.5: CPU Time for nearest neighbor search on uniform data

4.6 Performance Evaluation

Experiments have been done to show that the new nearest neighbor search algorithm outperforms the original nearest neighbor searching algorithm. We have implemented both R-Tree and R*-Tree in C under UNIX on a Sun Sparc computer, and have used our implementation in a series of performance tests whose purpose was to evaluate efficiency during searching by using both algorithms. Both the original and the improved nearest neighbor search algorithm have been used in these experiments. Experiments have been measured by CPU time used on nearest neighbor search and dimension of data.

Both uniform and clustered data have been used in experiments. The dimension of data varies from 4 to 32, and the number of data varies from 1000 to 40000. Figures 4.5 and 4.6 present the result of experiments on uniform data, while Figures 4.7 and 4.8 present the results of experiments on cluster data.

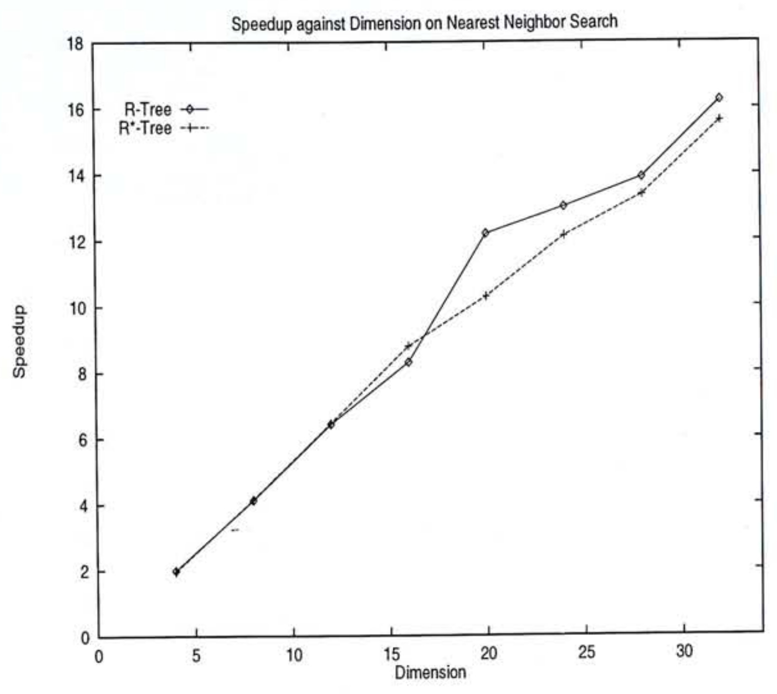


Figure 4.6: Speedup for nearest neighbor search on uniform data

R-Tree has dimensionality curses problem. That is, its performance will be dropped rapidly when it is dealing with high dimension data. It is because high dimension data introduce overlap to R-Tree. Therefore, the number of node access and the time for the nearest neighbor search will be increased. Figures 4.5 and 4.7 show that the number of CPU time used for R-Tree on nearest neighbor search will be increased with increasing rate.

R*-Tree also has the dimensionality curses problem even though R*-Tree is the best variant of R-Tree. However, as shown in Figures 4.5 and 4.7, R*-Tree used less CPU time on nearest neighbor search. Its performance is dropped when the number of dimension of data increased, but it still has better performance than R-Tree.

Both R-Tree and R*-Tree will gain when they use the new nearest neighbor search algorithm instead of the original one. Figures 4.5 and 4.7 show that CPU time used for both R-Tree and R*-Tree using the improved algorithm are smaller

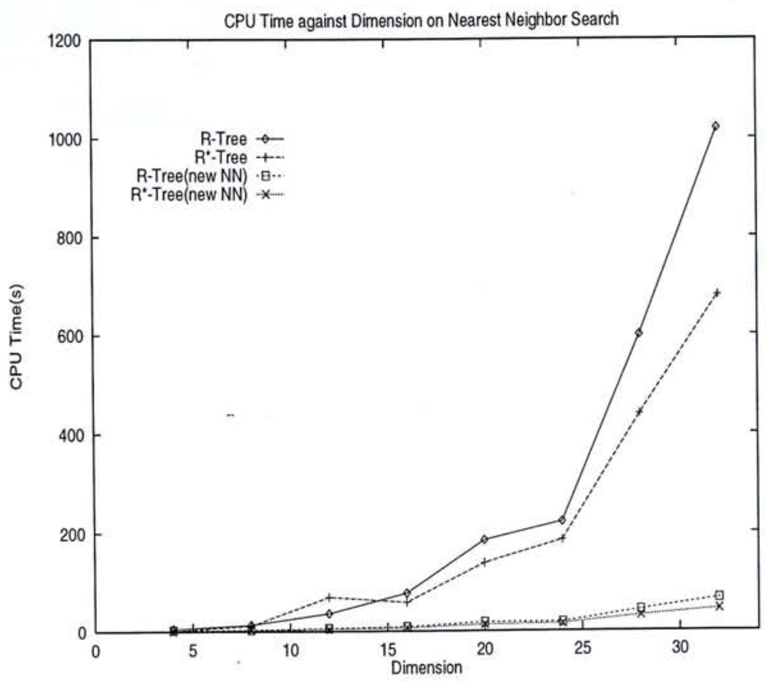


Figure 4.7: CPU Time for nearest neighbor search on clustered data

compared to the results of using the original algorithms, and the gains will be increased with the number of dimension. There are two reasons for the new algorithm outperforming the old one. First of all, the new algorithm does not need to calculate *MINMAXDIST* which is a very time-consuming processing. Furthermore, one less pruning heuristic is used to save time on processing.

Both Figures 4.5 and 4.7 show that the performances of nearest neighbor search on both R-Tree and R*-Tree drop rapidly as the number of dimension grows. However, when the improved nearest neighbor search on both R-Tree and R*-Tree is used, the processing time is shortened and its performance drops at a slower rate. Figures 4.8 and 4.6 show that the speedup factor of the improved nearest neighbor search algorithm increase with the number of dimension of data.

Experiments have also been performed on real data. The number of data varies from 10000 to 40000 and the number of dimension varies from 2 to 16. Figure

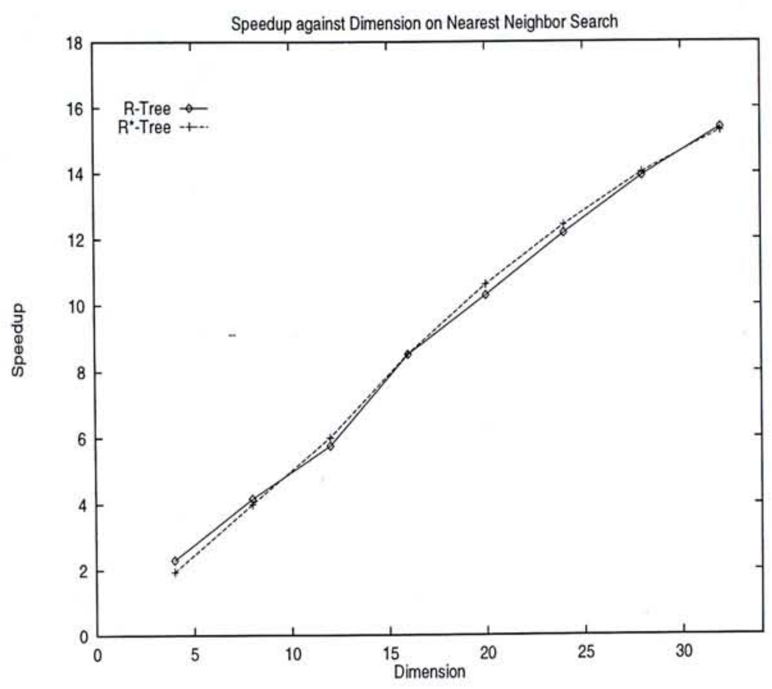


Figure 4.8: Speedup for nearest neighbor search on clustered data

4.9 shows the result of CPU time used on nearest neighbor search. Figure 4.10 shows the speedup factor of the improved algorithm on real data. The results of real data is similar to those of uniform data and clustered data. Since the real data have a lot of overlapping, the performance of R*-Tree cannot outperform R-Tree. When low dimension data are used, R*-Tree gives a substantial improvement to R-Tree. However, when the number of dimension increases, its performance is similar to R-Tree that it drops rapidly, and CPU time used on searching is almost the same for both trees. It is because there are a lot of overlapping when it is dealing with high dimension data. Its split and reinsert procedures hardly reduce any overlap, therefore, most of the nodes of the tree will be accessed. The figure shows the performance of both R-Tree and R*-Tree will be the same.

Figure 4.10 shows that the speedup factor for real data is higher than that of uniform data or clustered data. It is because the presence of overlapping makes

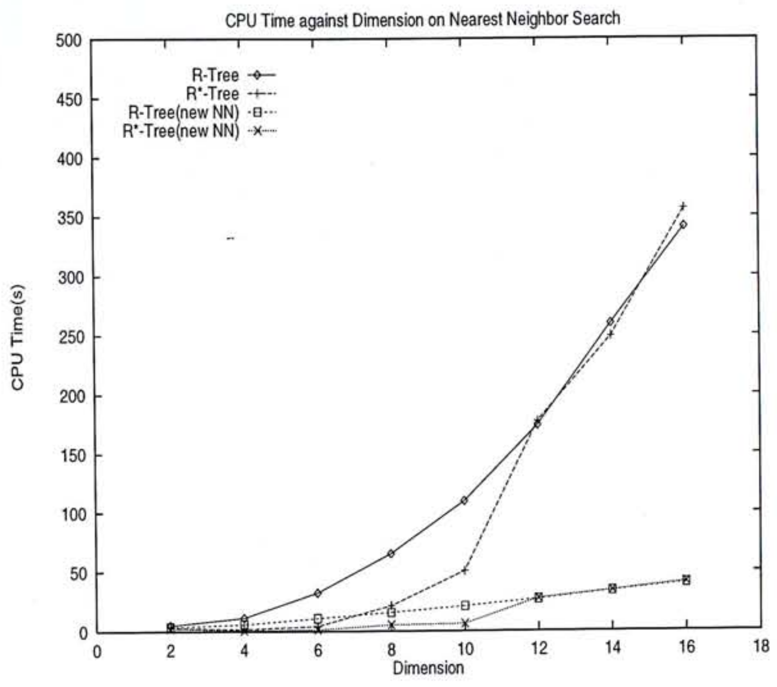


Figure 4.9: CPU Time for nearest neighbor search on real data

the number of node access on nearest neighbor increases. The original search algorithm has a lot of calculations of *MINMAXDIST* and the additional pruning operation. Therefore, the improved search algorithm can reduce more CPU time overhead and the speedup factor can be increased.

The improved N-nearest neighbor search algorithm has also been tested. Both the original and the improved N-nearest neighbor search algorithms are used. Experiments are to measure the CPU time used on nearest neighbor search against the dimensions of the data. Figure 4.11 shows the CPU time used on 10-nearest neighbor search on real data. Figure 4.12 shows the speedup of CPU used on 10-nearest neighbor search when the improved algorithm is used. The results are similar to the results in nearest neighbor search. Similarly, the performance of R*-Tree cannot outperform R-Tree. When the dimension of data increases, the performance drops rapidly, and CPU time used on searching is

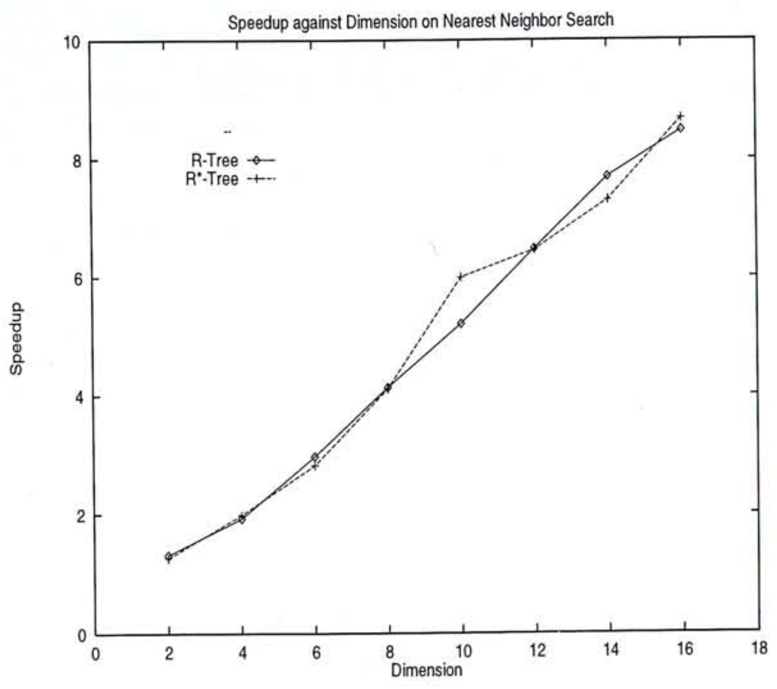


Figure 4.10: Speedup for nearest neighbor search on real data

almost the same for both trees. Therefore, the performance of R-Tree and R*-Tree is very similar when the same algorithm of search is performed. For the same tree structure, however, the improved N-nearest neighbor search algorithm is much better than the original N-nearest neighbor search algorithm. Figure 4.12 shows the difference between two algorithms on the same tree structure.

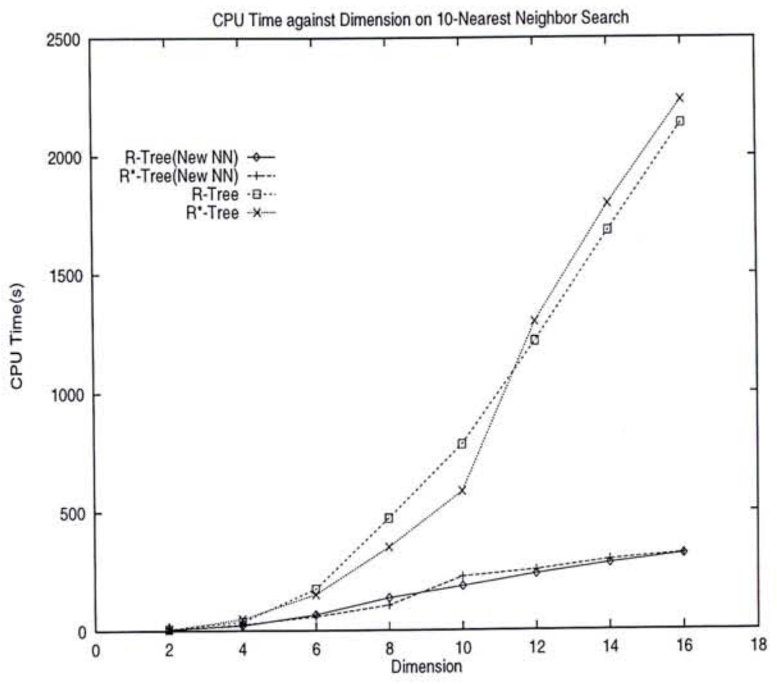


Figure 4.11: CPU Time for 10-nearest neighbor search on real data

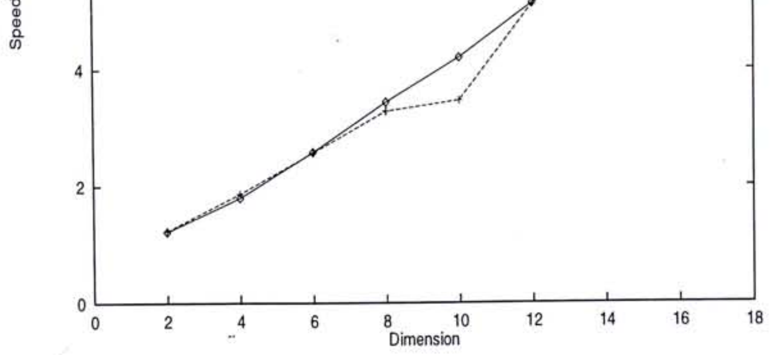


Figure 4.12: Speedup for 10-nearest neighbor search on real data

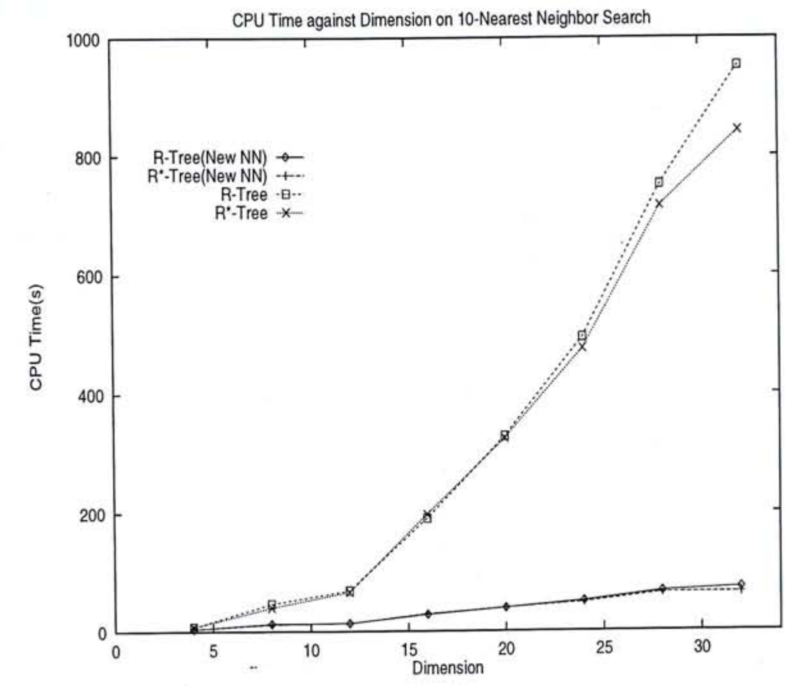


Figure 4.13: CPU Time for 10-nearest neighbor search on clustered data

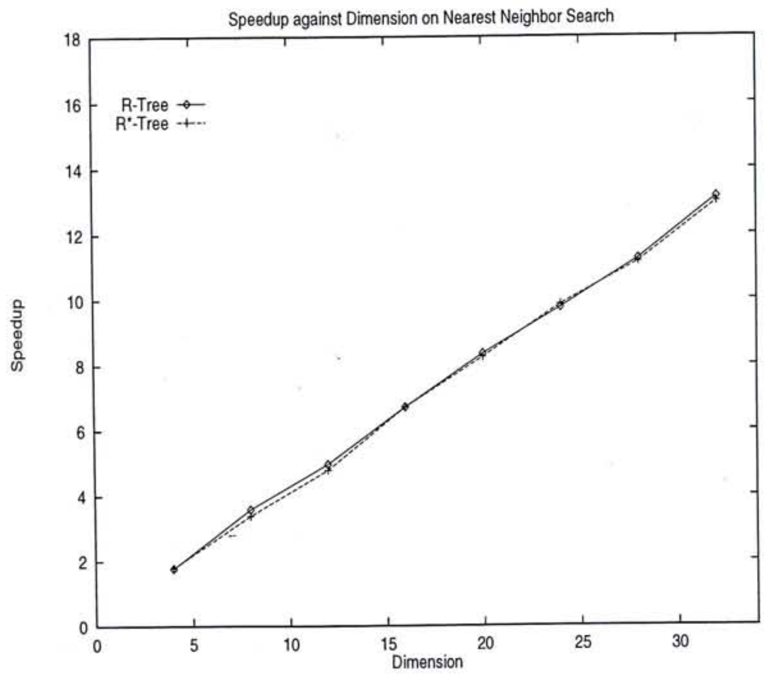


Figure 4.14: Speedup for 10-nearest neighbor search on clustered data

Chapter 5

Overlapping Nodes in R-Tree and R*-Tree

The main purpose of constructing index tree is providing an efficient method for information retrieval. Content-based index structures handle multidimensional data and they need to give a fast response to queries given by users. Since multidimensional data are usually large in size, efficiency of its index structure is very important. However, it is found that the performance of those content-based index structures are not satisfactory enough that, in worst case, the whole index tree will be accessed.

In this Chapter, the main cause of the poor performance of R-Tree and R*-Tree will be addressed. Overlapping, which is a common feature of R-Tree and R*-Tree, will be described. An overlap-free index tree may not outperform overlapping R*-Tree. For example, R^+-tree is an overlap-free index tree and it tries to provide overlap-free split which may make those nodes become less quadratic. Therefore, their margin will be increased and its performance on searching is worsen. However, overlapping nodes are still the main reason of the poor searching performance in content-based indexing. The presence of overlapping induces backtracking in searching which makes R-Tree and R*-Tree perform badly. How the presence of overlapping affects the performance of R-Tree and R*-Tree on both exact search and nearest neighbor search will be briefly explained.

5.1 Overlapping Nodes

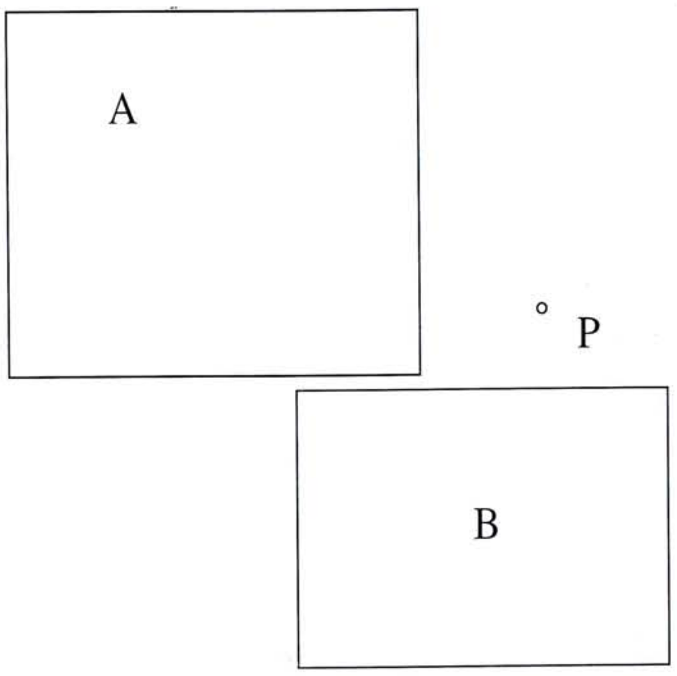


Figure 5.1: Two disjoint bounding boxes before the insertion

Overlapping of two nodes means that their minimum bounding boxes are not disjoint and they have common region. Figure 5.2 shows two nodes A and B overlap with each other and they have a common region C. On the other hand, if two nodes do not overlapping with each other, or they are called disjoint, they do not have a common region. Figure 5.1 shows two nodes A and B that do not overlap with each other.

B-Tree, which is a unidimensional index structure, does not have overlapping

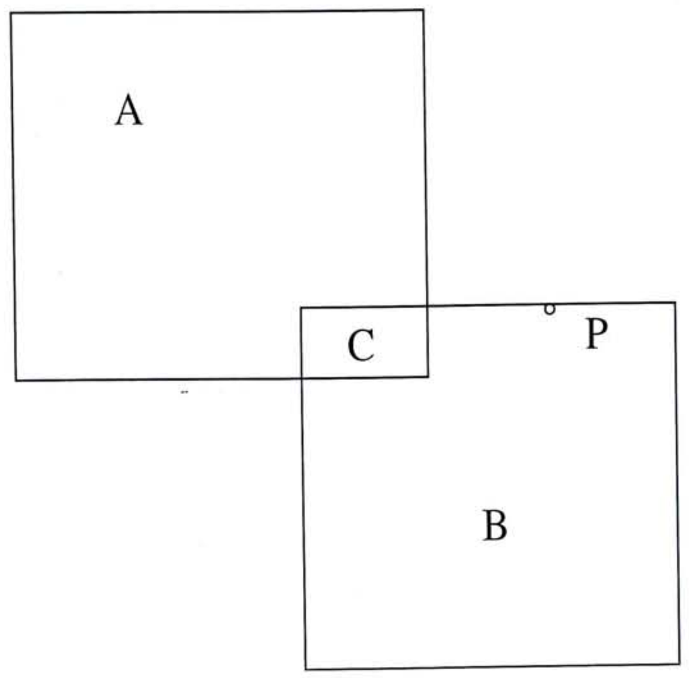


Figure 5.2: Two bounding boxes overlap with each other after the insertion

nodes. It is because the index structure handles simple one dimension data. Furthermore, a simple $<$ operator is used for inserting data or searching query so that disjoint nodes must remain disjoint. On the contrary, multidimension data make their index structures become complicated. In both R-Tree and R*-Tree, overlapping is not prohibited though it is not desired as well. R*-Tree tries to minimize overlapping, but, it cannot guarantee that overlapping can be avoided. In fact, R*-Tree is found to have a lot of overlapping nodes when they are dealing with high dimension data.

In R-Tree, overlapping can be introduced when an object is being inserted. Initially, no overlapping exists in the tree. However, it is possible that overlapping starts to occur by inserting an object to a node which is enlarged to overlap with another node. Figure 5.1 shows an example to illustrate it. Two nodes A and B do not have any overlapping initially. An object P is going to be inserted into the index tree. By the insert algorithm of R-Tree, since B has less area

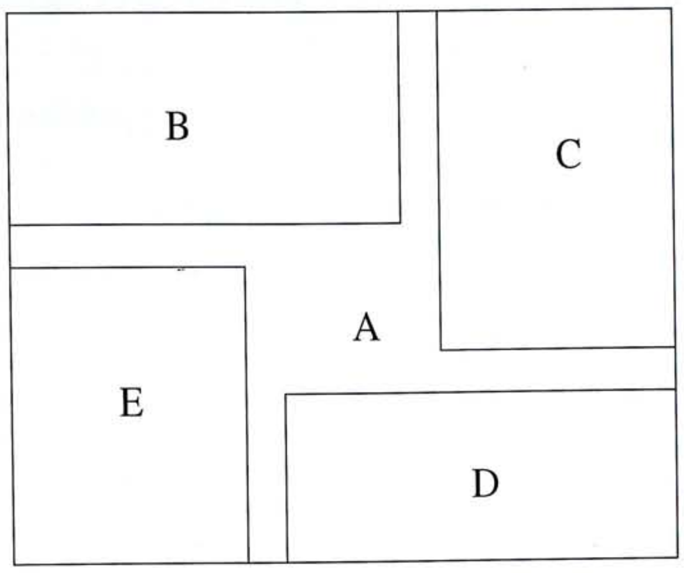


Figure 5.3: Overlapping nodes created during splitting

enlargement than A if P is inserted into it, P will be inserted to B and B will adjust its minimum bounding box so that the box encloses the object P. Figure 5.2 shows the result after the insertion. Now, nodes A and B overlap with each other. Although R*-Tree tries to minimize overlapping, overlapping still exist and R*-Tree cannot get rid of the problem.

Overflowing of a node may produce two overlapping nodes. Figure 5.3 shows an example of overlapping nodes caused by splitting of a node when overflow occurs. There is a node A which have four child nodes B, C, D and E. The split algorithms of both R-Tree and R*-Tree cannot produce two overlap-free nodes because splitting on either dimension axis cannot avoid overlapping. As the dimension of data becomes higher, the risk of overlapping becomes higher too.

5.2 Problem Induced By Overlapping Nodes

5.2.1 Backtracking

Backtracking means that the search process involves more than one node in the same level to be accessed. It will occur when the target is not found in the searched path and they want to search along another subtree. The reason for possible backtracking to occur is that there exists more than one search path when searching for an object. When a possible search path has been gone through, it may be necessary to go through another one and backtracking is thus occurred.

It is clear that backtracking makes the search process become inefficient because more nodes will be accessed. Backtracking does not exist in exact search of B-Tree because the search process is deterministic. At every stage of the search, an accurate search path will be determined. No matter the query object is present or not, the search process will be terminated after the leaf is accessed. On the other hand, the search in R-Tree and R*-Tree is not deterministic. The presence of overlapping nodes in the tree is a reason for backtracking to occur.

5.2.2 Inefficient Exact Search

The search algorithms of R-Tree are not efficient enough that, in worst case, it may access all nodes in the index tree. In this section, exact search algorithm of R-Tree is focused and its inefficiency is to be explained.

Since R-Tree allows overlapping bounding boxes, it cannot guarantee that a unique search path for an exact search query exists, and the performance will be degraded accordingly because of the frequently occurrence of backtracking. The following example can illustrate why backtracking is likely to occur when

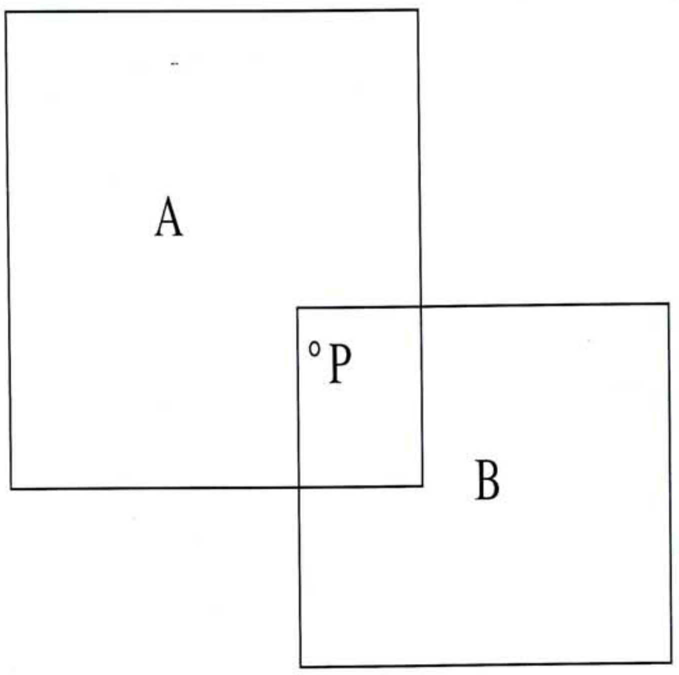


Figure 5.4: Two overlapping bounding boxes enclose the same object

overlapping regions present.

Suppose an object P is going to be searched. After certain steps have been carried out, node C is being accessed. There are two nodes A and B which are child nodes of C and their bounding boxes overlap with each other. It is also found that both of them enclose the query object P. The situation is shown in Figure 5.4 and its corresponding R-Tree structure is shown in Figure 5.5. At this point, we have to decide which node, A or B, should be accessed in order to find the query object. Although two bounding boxes enclose the query object P, it has one parent only, say B. That means, P can be found when we search along the subtree of node B, but it cannot be found if we search along the subtree of node A even though the bounding box of node A encloses P.

Although B is the parent of P, the fact is not known until both A and B have been accessed. Before A and B are searched, all can be known is that both of them may be the real parent of the query object P. Since the bounding box of

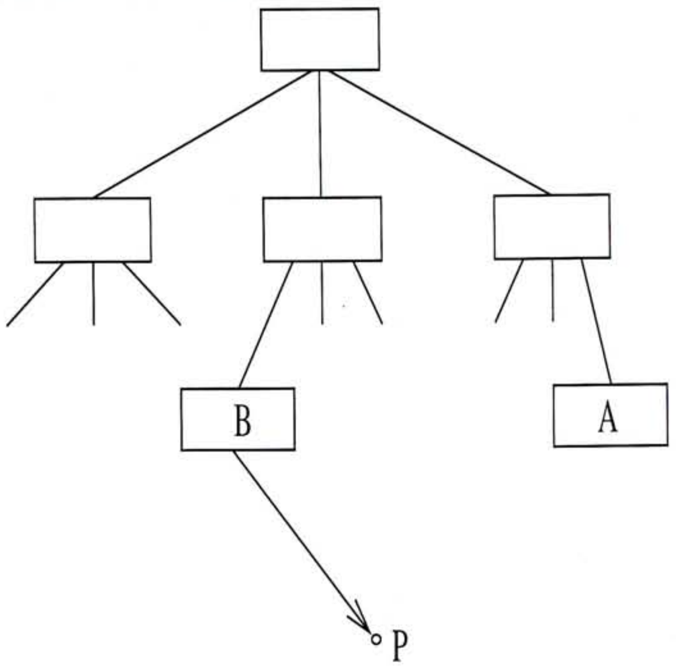


Figure 5.5: An example of R-Tree structure if overlapping happened

A encloses P, we can proceed the searching along the subtree of A. After the subtree of node A have been accessed and the target object is not found, it can be concluded that it should be located somewhere else in the tree, if it does exist. Therefore, backtracking exists to search along another subtree so as to get the target. After the searching process have been backtracked to node C and it is looking for another possible search path, it is found that the bounding box of node B does enclose the query object. Then, the subtree of B is being searched. Since the query object P located in the subtree of node B, it can be found at the leaf level of the subtree and the searching process come to an end.

In real cases, overlapping nodes in R-Tree and R*-Tree are commonly found, and multiple overlapping nodes will also be present. As we have shown in the above example, additional work on searching along subtree of node A is useless, but, it cannot be entirely eliminated. Therefore, the non-deterministic behavior in searching often causes the bad performance of R-Tree and R*-Tree.

5.2.3 Inefficient Nearest Neighbor Search

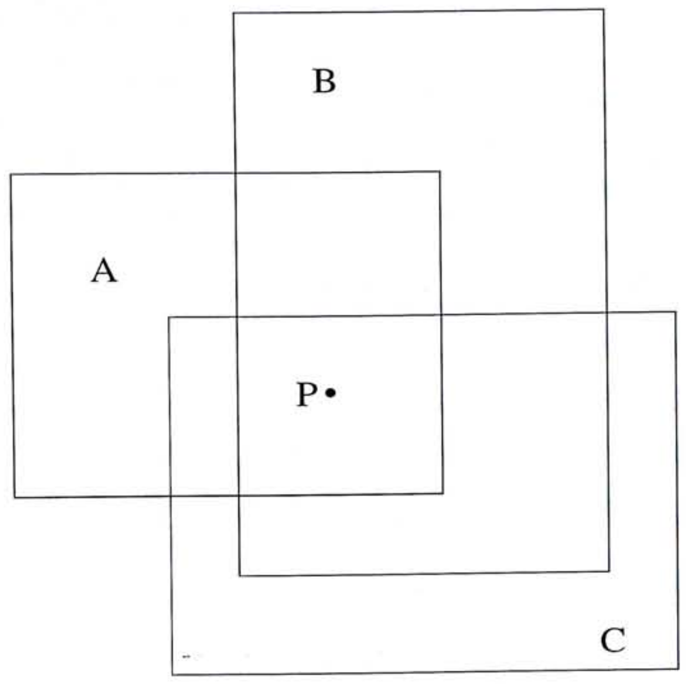


Figure 5.6: Three bounding boxes overlap with the same object

In this section, nearest neighbor search algorithm of R-Tree is focused and its inefficiency is to be explained.

The performance of R-Tree on nearest neighbor search is worse than that on exact match. As similar as exact search, backtracking is one of the reason that make nearest neighbor search in R-Tree become inefficient. Another reason is that even the real nearest neighbor of the query is obtained, it is not easy to identify that it is the desired one. N. Roussopoulos, S. Kelley, and F. Vincent[6] has designed an efficient nearest neighbor search algorithm on R-Tree and its variant, and it has been explained in Chapter 3. They provide two metrics for ordering in nearest neighbor search: *MINDIST* and *MINMAXDIST*. The use of these two metrics is to prune all nodes which are impossible to contain the nearest neighbor. Therefore, based on these two metrics, a set of nodes which may contain the nearest neighbor can be obtained. There is a substantial difference between nearest neighbor search and exact search. In exact search, we know the object being examine is the desired one or not. If we know that the object is what we want, the search process will be terminated. On the contrary, nearest neighbor search is not as deterministic as exact search. We want a variable to keep the temporary nearest neighbor. We will update the variable if we find another object which is closer to the query. However, we cannot guarantee that the one we got is the *real* nearest neighbor of the given query until all entries in the set have been searched. In fact, we have to do the same procedure in nearest neighbor search for the whole R-Tree.

The metrics, *MINDIST* and *MINMAXDIST*, are good approximations to the actual distance between object and query. However, it is no more a good approximation in dense overlapping area. First of all, there will be a lot of potential nearest neighbors of a given query, and they may be distributed into many subtrees. It implies that we have to search all such nodes in order to get the desired object, and we can determine that it is the real nearest neighbor of the query only after all those nodes are accessed. We can see that a lot of backtracking and unnecessary node accesses on R-Tree will occur for a nearest neighbor search query, and the situation is worse than that in exact search.

An example is given to illustrate the problem, the situation is shown in Figure 5.6 and its R-Tree structure is shown in Figure 5.6. Assume we are now trying to find the nearest neighbor of a given query point P. A, B and C are nodes of the R-Tree which are found that their bounding boxes enclose P. Therefore, *MINDIST* values of these three nodes are zero. Without loss of generality, assume that the searching order of these three nodes in the R-Tree is A, B and then C. After we have searched subtree of A, we get a temporary nearest neighbor a_i with $NN\_DIST_{a_i} \geq 0$. There are two cases to be occurred.

The first case is that the temporary nearest neighbor a_i is not the real nearest neighbor to the query. Since *MINDIST* of A, B and C are equal to zero, the condition $NN\_DIST_{a_i} < MINDIST$ cannot be satisfied. Heuristic 5 cannot be applied to prune node B and C from the Active Branch List, and hence the subtrees of B and C have to be searched. They have to be searched because either b_j or c_k, which are temporary nearest neighbor of the query under subtrees of B and C respectively, is the real nearest neighbor of P. In this case, it is similar to the situation of exact search that after we have reached the leaf of the tree and we find that the one we got is not the one we want, backtracking is necessary to help us find the required object.

The second case is that the temporary nearest neighbor a_i is the real nearest neighbor. Even if a_i is the real nearest neighbor, we cannot ensure whether it is the required object because it is possible that there exists an object in node B or node C which is the real nearest neighbor. After b_j and c_k have been found and their distances from the query are compared with that of a_i, we then know that a_i is the one we wanted. Therefore, no matter whether a_i, b_j or c_k is the real nearest neighbor to the query, backtracking does exist in this example, and subtrees which do not contain the nearest neighbor have also been accessed.

The above example is a case that shows inefficient search on R-Tree. The poor performance is not limited to a tree which have overlapping nodes that contain the query. For example, if a node D which is at the same level as A, B and C, and it does not contain query P, that is $MINDIST_D > 0$. It is possible that $\min(NN\_DIST_{a_i}, NN\_DIST_{b_j}, NN\_DIST_{c_k}) > MINDIST_D$, and subtree of node D is then needed to be searched too as $NN\_DIST_{d_l}$, which is the distance from the nearest object in node D to the query, is smaller than $\min(NN\_DIST_{a_i}, NN\_DIST_{b_j}, NN\_DIST_{c_k})$.

We have just shown the inefficiency of nearest neighbor in R-Tree and its variant. In fact, we cannot expect that searching an object on a multidimensional index structure is as efficient as that on traditional index structure like B-Tree. Pruning is necessary for nearest neighbor search. How to prune irrelevant objects from the searching list is very important, especially when overlapping cases occurred.

Chapter 6

Redundancy On R-Tree

Real multi-media applications find that existing index structures cannot give a satisfactory performance on data retrieval. Those applications handles complicated high dimensional data which need multidimensional index structures. In Chapter 2, the most popular index structures have been briefly described. In Chapter 5, how overlapping nodes in R-Tree and R*-Tree affect the performance of searching has also been explained.

In this Chapter, we focus on improving performance on data retrieval. The proposed solution is based on introducing redundancy on an index tree. A new multidimensional index structure Redundant R-Tree, which applies the idea of adding redundancy on the index tree, will be introduced. The construction method and the properties of the tree will also be described.

6.1 Motivation

As we have mentioned in Chapter 5, the presence of overlapping node in R-Tree induces backtracking in searching which worsens the performance. On the other hand, R^+-Tree, which is an overlap-free index structures, however, does not

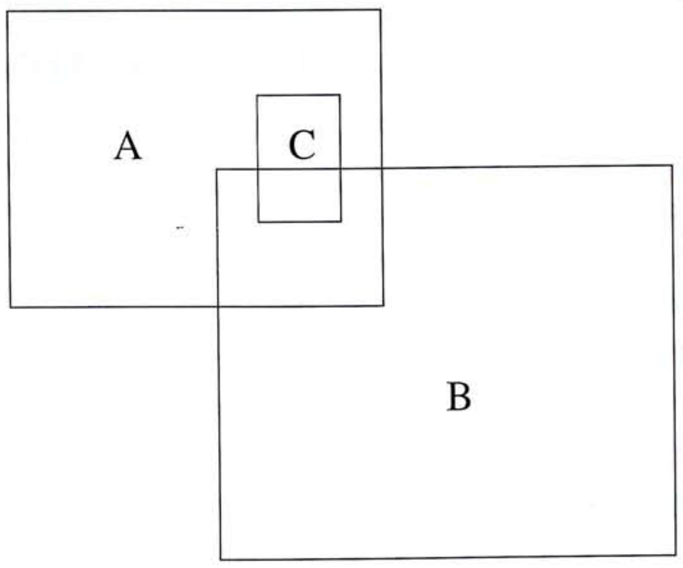


Figure 6.1: Two bounding boxes overlap with the same object

give a satisfactory performance. In fact, minimizing overlapping is not the only criterion for a content-based index structure to give a good performance. N. Beckmann *et al* in [7] specified several optimization criteria, and they pointed out that it is hard to optimize all of them in the same time. In fact, R*-Tree, which is designed by them, tried to optimize those criteria, and many experiments showed that R*-Tree gives the best performance. Nevertheless, R*-Tree still suffers from the dimensionality curse problem that its performance drops when the dimensionality of data is increased. It is because R*-Tree has a lot of overlapping nodes when those data being dealt with induce backtracking when a search query is being performed. Therefore, the performance can be improved if backtracking can be minimized or even eliminated and it is the task of adding redundancy on an index tree.

6.2 Adding Redundancy on Index Tree

The concept of adding redundancy to an index tree is that, every node N should contains all nodes n_i if and only if MBR_N encloses n_i, and there exist not a n_j

which is not contained by N but enclosed by MBR_N. An example is given to clarify the concept. In Figure 6.2, both bounding boxes of node A and B enclose an object C. In R-Tree and R*-Tree, either A or B contain C. When redundancy is added to index tree, both node A and B should contain C. In this case, if an exact search query is given to find the object C, a unique search path for the query can be provided. No matter node A or B is accessed first, the object must be found without backtracking. In case the object is not found, it can be ensured that the object does not exist in the index tree and there is no need to backtrack other subtrees. In nearest neighbor search, backtracking can be reduced. If the query object is contained by bounding box of node A, MBR_A, the subtree of node A *probably* contains the real nearest neighbor. The special case is when the query located very near to the boundary of the bounding box, the real nearest neighbor may not be enclosed in the bounding box. However, in the original R-tree nearest neighbor search algorithm, real nearest neighbor may not be obtained, or at least may not be recognized as the real nearest neighbor even though the query located at the center of the bounding box until backtracking is carried out to access other subtrees. It is obvious that performance have been improved after redundancy is added to an index tree as it is expected to have less node access in both exact and nearest neighbor search than that of original R-Tree.

6.3 R-Tree with Redundancy

6.3.1 Previous Models of R-Tree with Redundancy

In order to add redundancy on an index tree, the structure of the tree should be modified. Before our proposed index tree structure is introduced, we will describe previous index tree structures with redundancy. The advantages and disadvantages will also be discussed explaining why and how our proposed index

Algorithm 6.1 *OverlapChild*

```
Procedure OverlapChild($N_1$, $N_2$)
Input : $N_1$
        /* Node to be inserted by redundant entry */
        $N_2$
        /* Redundant entry */
Begin
    If $N_2$ is a leaf
      If OverlapTest($N_1$, $N_2$) == TRUE
        Return TRUE
      Else
          Return FALSE
    Else
        /* to test if any child of $N_1$ contains $N_2$ */
        For all children $n$ of $N_1$
            If OverlapChild($n$, $N_2$) == TRUE
              Return TRUE

    Return FALSE
End
```

tree structure is constructed.

In the first model, every non-leaf node has two kind of entries: normal child pointers as the same as those in R-Tree, and a list of redundant entries. Let us denote the model by $Model_A$.

First of all, inserts all the data points the same way as we do to the R-Tree. After that, starting from the lowest level of tree node, we try to add redundant entries to every node. We test overlapping of the nodes on the same level. After all nodes on one level have been tested, we move upwards to test overlapping of their parent nodes until root node is reached. We illustrate the algorithm by giving an example: A has children B and C, D has children E, F and G, where A and D are on the same level i, and, $B, C, E, FandG$ are on the same level

Algorithm 6.2 *AddOverlap*

```
Procedure AddOverlap($N_1$, $N_2$)
Input : $N_1$
        /* Node to be inserted by redundant entry */
        $N_2$
        /* Redundant entry */
Begin
    If $N_2$ is not a leaf
      For all children $n$ of $N_2$
          /* to test whether they have overlapping region */
          If OverlapTest($N_1$, $n$) == TRUE
            /* add redundant entries from $n$ to $N_1$ */
            Call AddOverlap($N_1$, $n$)
    Else
      If $N_2$ is contained by $N_1$
        /* test if any child of $N_1$ contains $N_2$ already */
        If OverlapChild($N_1$, $N_2$) == FALSE
          Add $N_2$ to the redundant list of $N_1$.
End
```

$i+1$ too, and they are not leaf nodes. Assume we are now testing overlapping on level $i+1$. We test pairs of nodes on level $i+1$ to see if they are overlapping with each other. For example, C and E are under testing now, and if they do overlap with each other, we will add all leaf nodes enclosed by $C(E)$ to $E(C)$ as its redundant entries. After all pairs are examined, we will test all entries on level i and A, B will be tested. In order to reduce unnecessary redundancy, we enfore that parent node does not have any redundant entries that appear in its children. That means, A will not get a data point p as its redundant entries if either B, C or any their descendents get p. The advantage of this model is that it provides an easy method to add redundancy to the index tree. The disadvantage of this model is that the index tree is not height-balanced which cannot give a fair performance on searching different objects.

Algorithm 6.3 *StartOverlap*

```
Procedure StartOverlap($N_1$, $N_2$)
Begin
      Set N to be the pointer to the lowest level of Level List
      If N $\neq$ NULL
         Set $L_1$ to be the head of N
         If $L_1 \neq$ NULL
            Set $L_2$ to be the one next to $L_1$
            If $L_2 \neq$ NULL
               If $OverlapTest(L_1, L_2) ==$ TRUE
                  Call $AddOverlap(L_1, L_2)$
                  Call $AddOverlap(L_2, L_1)$
               Advance $L_2$ to the next entry
            Advance $L_1$ to the next entry
         Advance $N$ upwards on the Level List
      Return
End
```

The second model is a modification of the first model. The first model has a problem that the data objects are essentially on the same level. Some of them may be accessed at root node while some of them may be accessed at the leaf level which are the same as R-Tree. Let us denote the model by $Model_B$.

The algorithm of adding redundancy to the index tree is similar to the first model. The only difference is how to add a redundant entry to a node. In the first model, a list which store redundancy entries will be created. In this model, redundant entries will make up a tree such that the resultant index tree is height-balanced. The advantage of this model is that the tree is height-balanced which gives a fair performance on searching of different object. The disadvantage of this model is that it is difficult to avoid multiple access to the same node in a search query. For example, in Figure 6.2, both node A and B have a redundant entry C. If C has been accessed and now node B is being accessed, it is difficult to prevent from accessing node C again in the same query.

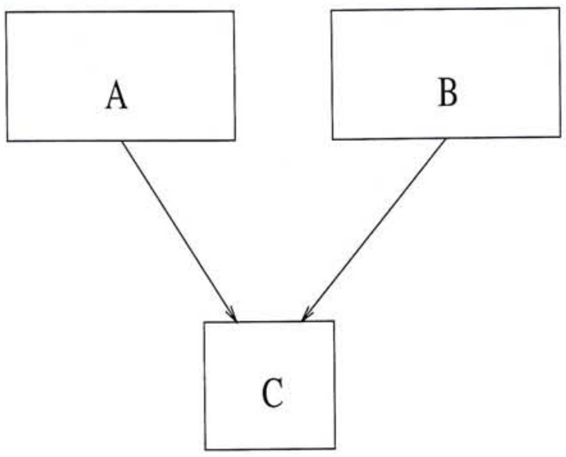


Figure 6.2: Example of R-Tree: Bounding box

6.3.2 Redundant R-Tree

In previous section, different models that apply redundancy on R-Tree has been described. Their advantages and disadvantages have also been mentioned. A finalized index structure for adding redundancy on R-Tree is now to be presented.

Our proposed new index structure which is called Redundant R-Tree is a R-tree with redundancy. That means Redundant R-Tree is a variant of R-Tree. The formation of a Redundant R-Tree is the same as that of a R-Tree except it will be augmented with redundant entries. Its construction method as well as its properties will also be described.

Our index structure will be based on $Model_B$. As mentioned in the previous subsection, it has a drawback of using a lot of storage to store those redundant entries. Furthermore, it is difficult to prevent multiple access to the same node in the same query. In order to solve the problem, we design our tree structure as in Figure 6.2. In the example, node A and node B share the same redundant entry C. In this case, it is obvious that the demand of storage will be decreased. In addition, it is easier to avoid multiple access to the same node in the same query if this structure is being used.

6.3.3 Level List

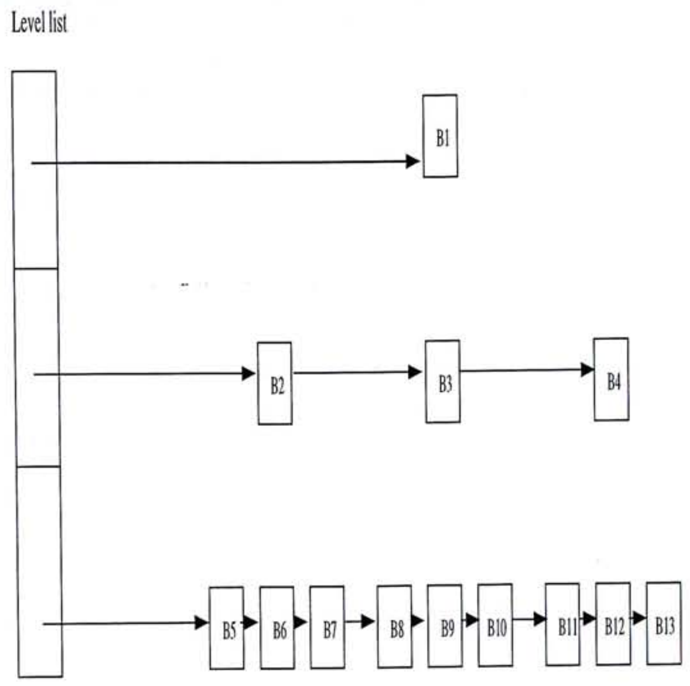


Figure 6.3: Level List

In our index structure, there is an array of level lists. A level list is a difference linked list which is used to link up all nodes of the tree which are in the same level. Every entry in the level list has a pointer which points to the first object at that level. For example, the i^{th} entry of the level list would point to the first object in level i, say A_{i_1}. A_{i_1} will in turn point to another node, say A_{i_2}. An example is given in Figure 6.3. Start from a level list, we can retrieve all nodes which are on the same level in the tree.

The reason for constructing such the list is that it will make adding redundancy on index tree easier. Assume redundant entries are being inserted into a node N_1. It is necessary to have an efficient way to find all N_2 which are redundant entries to N_1. Details about how to use the list will be described in the section of inserting redundancy to R-Tree.

6.3.4 Inserting Redundancy to R-Tree

Algorithm 6.4 *Overlap_Test*

```
Procedure Overlap_Test($N_1$, $N_2$)
Input : $N_1$
        /* Redundant node inserted to $N_1$ */
        $N_2$
        /* Redundant node $N_2$ */
Begin
    For i := 1 to no. of dimension

        Set $bl_{1_i}$ to be lower bound of $N_1$ on $i^{th}$ dimension
        Set $bu_{1_i}$ to be upper bound of $N_1$ on $i^{th}$ dimension
        Set $bl_{2_i}$ to be lower bound of $N_2$ on $i^{th}$ dimension
        Set $bu_{2_i}$ to be upper bound of $N_2$ on $i^{th}$ dimension

        If $bu_{1_i} < bl_{2_i}$ or $bu_{2_i} < bl_{1_i}$
            Return FALSE

    Return TRUE
End
```

A Redundant R-Tree is built in order to provide more efficient searching algorithms than those of R-Tree. The whole constructing process is started at building an R-Tree first. After we have built the R-Tree, we have to annotate the tree so that redundant entries will be added to nodes if necessary. The adding redundant entries started from the lowest level of the index tree. We pick every pair of nodes in the same level to see whether they overlap with each other. If it is the case, we will further test whether they overlap with another's child nodes, and we will add those child nodes to the node in which they do not belong to if they do have overlapping regions. The process is repeated until all pairs in the same level have been examined, and then we move upwards and the process is repeated again until the root of tree is reached. Actually, the index tree may be described as a directed graph, as a node may have more than one

Algorithm 6.5 *Overlap*

```
Procedure Overlap($N_1$, $N_2$)
Input : $N_1$
        /* Redundant node inserted to $N_1$ */
        $N_2$
        /* Redundant node $N_2$ */
Begin
    For i := 1 to no. of children of $N_2$

        Set $N_{2_i}$ to be the $i^{th}$ children of $N_2$

        Call $OverlapTest(N_1, N_{2_i})$

        If $N_1$ and $N_i$ overlaps with each other

            Insert $N_{2_i}$ to $N_1$

    Return
End
```

parent.

The algorithm of inserting the redundant entries to the index tree is now to be described. First of all, *OverlapStart*, which is shown in Algorithm 6.6, will be invoked after the R-tree has been constructed. *OverlapStart* is the main procedure to build the Redundant R-Tree. The procedure starts to add redundant entries at the lowest level of the index tree. It tests every pair of nodes in the same level whether those nodes overlap with each other or not. Here, the pair of nodes to be tested are retrieved through the use of level list, and the testing is done by invoking a procedure called *OverlapTest* which is shown in Algorithm 6.4. If the pair of nodes do overlap with each other, then *OverlapStart* tries to add their overlapped child nodes to these two nodes by invoking a procedure called *Overlap* which is shown in Algorithm 6.5.

Algorithm 6.6 *Overlap_Start*

```
Procedure Overlap_Start
Begin
        Set N to be the pointer to the lowest level of Level List
        If N $\neq$ NULL
           Set $L_1$ to be the head of N
           If $L_1 \neq$ NULL
              Set $L_2$ to be the one next to $L_1$
              If $L_2 \neq$ NULL
                 Call $OverlapTest(L_1, L_2)$
                 If L1 and L2 overlap with each other
                    Call $Overlap(L_1, L_2)$
                    Call $Overlap(L_2, L_1)$
                 Advance $L_2$ to the next entry
              Advance $L_1$ to the next entry
           Advance $N$ upwards on the Level List
        Return
End
```

If two nodes A and B are in the same level and they overlap with each other, the procedure *Overlap* will be invoked to add redundant entries to the overlapped nodes. In this procedure, if there is a child node a_i of A which overlaps with B, then node B will be updated so that the redundant entry a_i will be inserted to the node B. As the relation of overlapping is symmetric, the operation will be done again so that it adds redundant entries to node A if necessary. Therefore, similar to the previous operation, b_j, which is a child node of node B, will be inserted as redundant entry of node A if b_j overlaps with node A.

The procedure *OverlapTest* is used to test whether two bounding boxes overlap with each other. Two bounding boxes have overlapping regions if and only if their line segments overlap with each other in all dimensions. For example, node A and B are two n-dimensional bounding boxes. Let A_i and B_i be line segment of bounding boxes of node A and B in the i^{th} dimension respectively. Let A_{a_i}

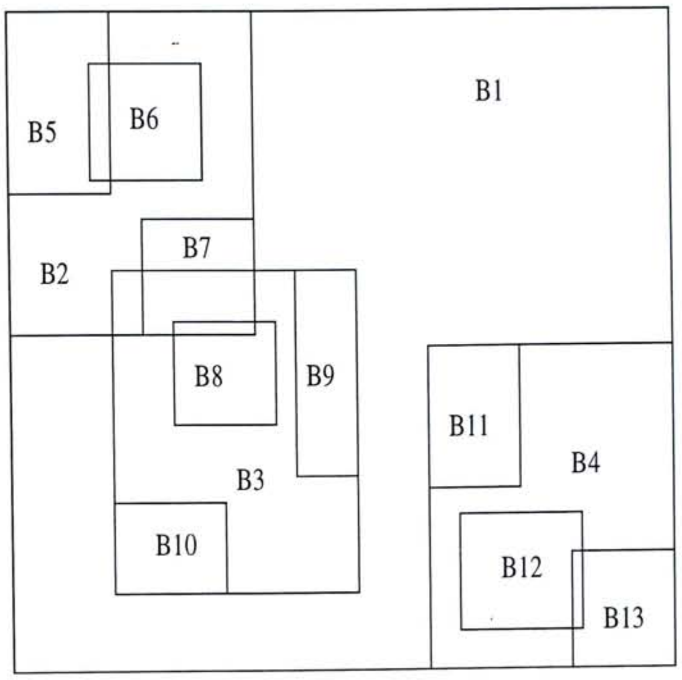


Figure 6.4: An example: overlapping bounding boxes

and A_{b_i} be the lower and upper limits of A_i, while B_{a_i} and B_{b_i} be the lower and upper limits of B_i. There are four cases to consider:

1. $A_{a_i} \leq B_{a_i} \leq A_{b_i} \leq B_{b_i}$;
2. $B_{a_i} \leq A_{a_i} \leq B_{b_i} \leq A_{b_i}$;
3. $A_{a_i} \leq B_{a_i} \leq B_{b_i} \leq A_{b_i}$;
4. $B_{a_i} \leq A_{a_i} \leq A_{b_i} \leq B_{b_i}$.

For all pairs of line segment A_i and B_i, if any one of these four cases are satisfied, we can conclude that bounding boxes of A and B overlaps with each other. Otherwise, if any pair of line segment A_i and B_i such that none of the above four cases is satisfied, we say that bounding box of A and B does not have overlapping region. *OverlapTest* in Algorithm 6.4 uses a simplified test: for any dimension, if the lower bound of a node is larger than the upper bound of another node, these two nodes must not have any overlapping region.

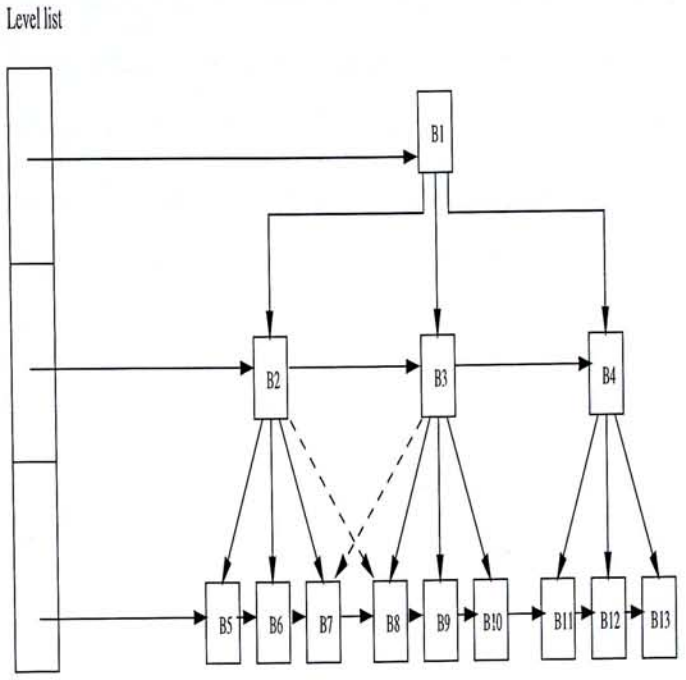


Figure 6.5: An example: The index tree structure

After all pairs of nodes in the same level have been tested, the process will be repeated at upper level, that is, testinging pairs of nodes in upper level. The whole process will be repeated until it is now at root level. The inserting redundancy process is then terminated and the Redundant R-Tree has been constructed.

An example is given to illustrate the process of inserting redundancy to an index tree. The example is shown in Figure 6.4. Three level lists: (B_1), $(B_2 \rightarrow B_3 \rightarrow B_4)$ and $(B_5 \rightarrow B_6 \rightarrow B_7 \rightarrow B_8 \rightarrow B_9 \rightarrow B_{10} \rightarrow B_{11} \rightarrow B_{12} \rightarrow B_{13})$. When *OverlapStart* is called, it starts to add redundancy at lowest level of the index tree. Assume it is to add redundancy to the node of B_2, B_3 and B_4. The first entry of the level list $(B_2 \rightarrow B_3 \rightarrow B_4)$, B_2, is L_1 in *OverlapStart*. Its next entry B_3 is L_2 in *OverlapStart*. Then the procedure tests whether these two nodes have overlapping regions. Since they overlap with each other, *Overlap* will be invoked to test if B_2 overlaps with any child node of B_3. B_8, B_9 and B_{10}

are children of B_2. B_9 and B_{10} do not overlap with B_2 but B_8 does. Therefore, B_8 will be inserted to B_2 as its redundant entry. Similarly, *Overlap* will be invoked to test if B_2 overlaps with any child node of B_3. B_5, B_6 and B_7 are children of B_1. B_5 and B_6 do not overlap with B_1 but B_7 does. Therefore, B_7 will be inserted to B_1 as its redundant entry. Then, L_2 is advanced to be B_4 now. Since it does not overlap with B_2 and it is the last entry in the level list, L_1 is advanced to be B_3 but L_2 remains to be B_4. Again, B_3 does not overlap with B_4. As all pairs of nodes in the same level have been tested, nodes in upper level will be tested. As the upper level list contains the root node only, the inserting redundancy terminates. The resultant Redundant R-Tree is shown in Figure 6.5.

6.3.5 Properties of Redundant R-Tree

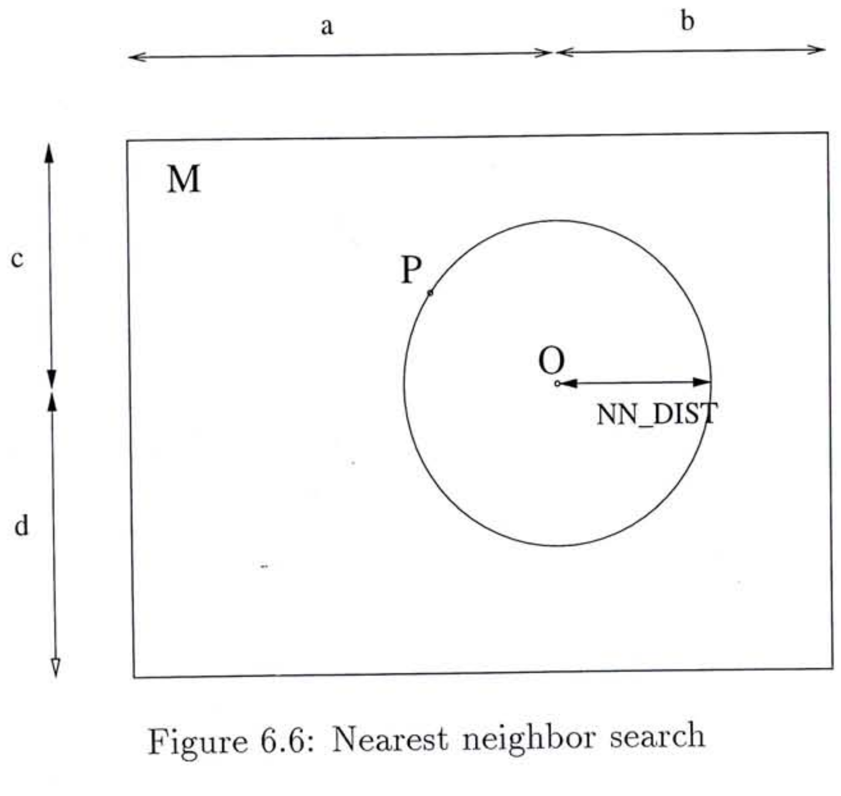


Figure 6.6: Nearest neighbor search

Redundant R-Tree is a variant of R-Tree and it inherits some properties of R-Tree which have been mentioned in Chapter 2. On the other hand, Redundant R-Tree is designed so that a better performance on data retrieval can be achieved, which implies that Redundant R-Tree has its own properties. Exact search and nearest neighbor search algorithms of Redundant R-Tree can be designed by making use of those properties.

The first property is about the inter-relation between bounding boxes and nodes.

Property 1 *Let A and B be nodes in a Redundant R-Tree at level i and i+1 respectively. The bounding box of A encloses or overlaps with the bounding box of B if and only if node A has an entry to node B.*

Theorem 4 *Property 1 is valid.*

Proof:

($\Leftarrow$):

If node A has an entry to node B, then node B is either a normal child node or a redundant entry of node A. If node B is a normal child node of node A, then, according to properties of R-Tree, it is enclosed by the bounding box of node A. If node B is a redundant entry of node A, $OverlapTest(A, B)$ must be evaluated to be true which determines node B is a redundant entry of node A. $OverlapTest(A, B)$ equals to true means that the bounding box of A encloses or overlaps with the bounding box of node B. Therefore, if node A has an entry to node B, then the bounding box of A encloses or overlaps with the bounding box of node B.

($\Rightarrow$):

If the bounding box of node A encloses or overlaps with the bounding box of node B, there are two possibilities. The first case is that node B is already a child node of node A, and node A is trivial to have an entry to node B. The second case is that B is node a child node of node A. Nodes A and B_{parent}

which is the parent of node B are in the same level. When $OverlapStart$ is being executed, $OverlapTest(A, B_{parent})$ will be evaluated. By properties of R-Tree, B_{parent} must encloses node B which implies that B_{parent} overlaps with node A and $OverlapTest(A, B_{parent})$ will be evaluated to be true. Then $OverlapTest(A, B)$ will be evaluated to be true also. Therefore, node B will be inserted to node A as the latter's redundant entry. Hence, the statement is correct.

As both statements ($\Leftarrow$) and ($\Rightarrow$) are proved to be correct, the theorem holds. $\square$

The second theorem is a generalization of Theorem 1.

Property 2 *Let node A be ancestor of another node B in Redundant R-Tree. If the bounding box of A encloses or overlaps with the bounding box of B, node A has a path to node B.*

Theorem 5 *Property 2 is valid.*

Proof: The proposition is to be proved by Mathematical Induction.

Let P(n) be the proposition of Property 2. Let n be the difference in level between node A and B.

When $n = 1$, the proposition is proved to be correct by Theorem 4. Assume the proposition is correct when $n = k$ where k is larger than or equal to one. When $n = k + 1$, let node C be the parent of node B such that the difference in level between node A and C is k. If the bounding box of node A encloses or overlaps with the bounding box of node B, by properties of R-Tree, the bounding box of node A encloses or overlaps with the bounding box of node C. As the difference in level between nodes A and C is k and the proposition holds when $n = k$, node A has a path to node C. Because node C is the parent of node B, node C has a path to node B, and consequently, node A has a path to node B. Therefore, the proposition is correct when $n = k + 1$ which means that it is also true for all positive integer k.

Thus, the proposition P(n) is proved to be correct by using Mathematical Induction. □

The converse of the statement is not necessarily true. That means, if a node A has a path to node B, then node B may not be enclosed or overlapped with the bounding box of node A. However, it may not be an undesirable property. Consider that a nearest neighbor query has been given which is located very near to the boundary of the bounding box. Its nearest neighbor may be located outside the bounding box of current node. If the converse of the statement is allowed, then the nearest neighbor may be accessed even the bounding box of the current node does not enclose or overlap with the nearest neighbor.

The following theorem tells a geometric property of bounding boxs in the index tree.

Property 3 *Let NN_DIST be the distance between temporary nearest neighbor and query. Let O=$(o_1, o_2, ..., o_m)$ be a query and P=$(p_1, p_2, ..., p_m)$ be the temporary nearest neighbor of an query O. Let M=(L, U) be the current bounding box where L=$(l_1, l_2, ..., l_m)$ is the lower bounds and U=$(u_1, u_2, ..., u_m)$ is the upper bounds of the bounding box respectively.*

If NN_DIST $\leq \min(|o_i - l_i|, \forall i = [1, 2, ..., m])$ and NN_DIST $\leq \min(|o_i - u_i|, \forall i = [1, 2, ..., m])$, then we can be sure that the nearest neighbor is located within the current bounding box of node M and it can be obtained after node M has been accessed.

Theorem 6 *Property 3 is correct.*

Proof: If $NN\_DIST \leq \min(|o_i - l_i|, \forall i = [1, 2, ..., m])$ and $NN\_DIST \leq \min(|p_i - u_i|, \forall i = [1, 2, ..., m])$, every object which locates outside the bounding box cannot be the nearest neighbor because the distance between the object and the query must be larger or equal to the minimum distance between the query and the boundaries of the bounding box, which is larger than or equal to the distance between the query and the temporary nearest neighbor. Furthermore, by Theorem 5, we guarantee that all objects which are enclosed by the same bounding box can be found in the same subtree. The real nearest neighbor, which has smaller or equal distance to the query than that of temporary one, must locate within the bounding box and the postconditions are satisfied. Therefore, the statement is proved to be correct. □

Figure 6.6 is an example to illustrate Theorem 6. Let M be a Redundant node in Redundant R-Tree. Let O be the query and P is the nearest neighbor in node M to the query, and the distance is NN_DIST. Let a, b, c, d be distances from the query to the boundaries of node M. If the precondition stated in the theorem is true, that is, $NN\_DIST \leq a$, $NN\_DIST \leq b$, $NN\_DIST \leq c$ and $NN\_DIST \leq d$ are satisfied, and then the nearest neighbor is located within the bounding box of node M. By Theorem 5, the nearest neighbor must be found under the subtree of node M. In case P is found to be the nearest neighbor in node M, then it is the real nearest neighbor to the query among all objects indexed by the index tree.

The theorem is very important to Redundant R-Tree because we can make use of it to design its own efficient searching algorithms so as to reduce the numbers of node accesses when search queries are being performed. The search algorithms will be discussed in Chapter 7.

Chapter 7

Searching in Redundant R-Tree

The construction method and some properties of Redundant R-Tree have been described in Chapter 6. Since there are differences between R-Tree and Redundant R-Tree, it is obvious that new searching algorithms should be designed for the new index structure so that the characteristics of the new index structure can be utilized and a better data retrieval performance can be achieved. In this Chapter, algorithms of exact-search and nearest neighbor search of Redundant R-Tree will be described. Examples will also be provided to demonstrate how these algorithms work.

7.1 Exact Search

In this section, a complete exact search algorithm of Redundant R-Tree will be given. Several examples will also be shown to demonstrate how the algorithm works.

In Redundant R-Tree, every bounding box contains all child nodes which are enclosed by the bounding box. That means, if we search a query points A which are contained by a bounding box B, we can find A by accessing the subtree

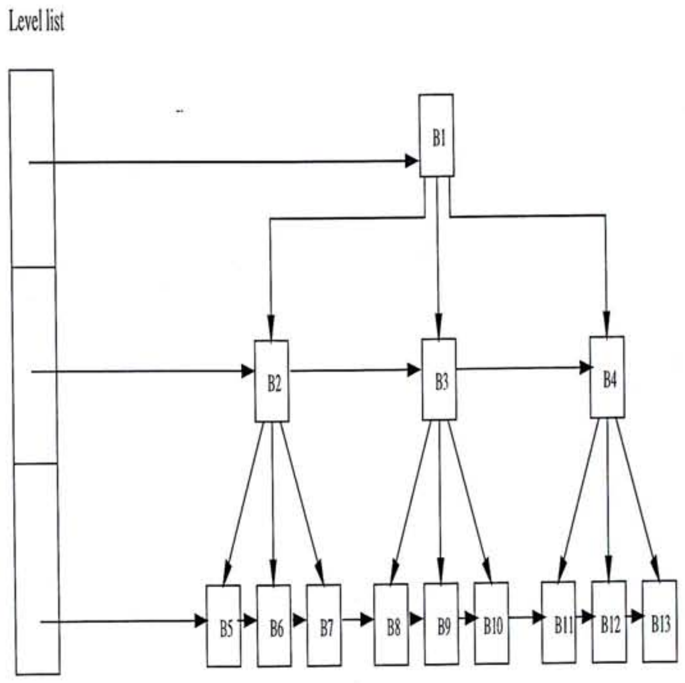


Figure 7.1: Tree structure of Redundant R-tree

of node B. Compared it to the original R-Tree algorithm in which we cannot guarantee that the subtree of node B is parent of A although the bounding box of node B encloses A. It may be the case that another bounding box C which overlaps with B and encloses A, and one of its subtree contains A. We have described the overlapping nodes in R-Tree in Chapter 5. After a Redundant R-Tree is constructed, exact search query can be performed more efficiently. The exact search algorithm is given in Algorithm 7.1.

The exact search algorithm is designed for Redundant R-Tree. Since Redundant R-Tree is a variant of R-Tree, its search algorithm is similar to the exact search algorithm of R-Tree. Again, the exact search algorithm of Redundant R-Tree is based on the containment test. That is, a node will be accessed if the query is enclosed by the bounding box of the node. Otherwise, it will be left unvisited. If there are more than one node which bounding boxes enclose the query, we just simply choose one node to visit. The process is repeated until we reach

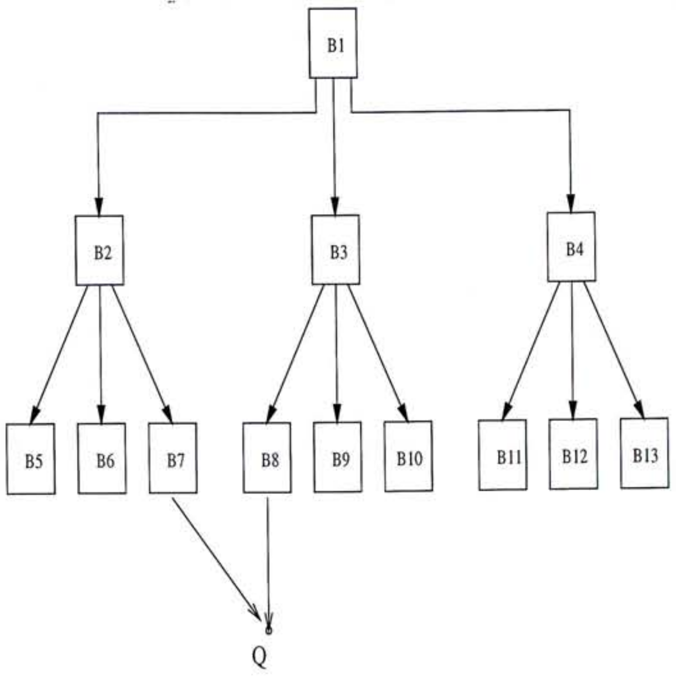


Figure 7.2: Exact search on Redundant R-Tree

the leaf level of the tree. Then we try to find the desired object. If the query object is found, then it reports that the object is found and the search process terminated. Otherwise, the search process will be continued until either the query is found or all objects of the node have been accessed. If it is the latter case, then it reports that the object is not found, and the search process will be terminated. No matter whether the object is found or not, no backtracking will be present. It is because, by the property of Redundant R-Tree, we know that the object is not present in the tree. It is very different to the original R-Tree exact search algorithm that the object may exist in somewhere of the tree which implies that the search process must proceed and backtracking takes place.

Recalls the example given in Figure 3.1. The Redundant R-Tree constructed in this example is shown in Figure 7.2. Now, an query Q is specified. The searching will be started at the root of the tree, B_1. As the bounding box of B_1 encloses the query, the searching will continue and all child nodes of B_1 will be

Algorithm 7.1 Exact search algorithm for Redundant R-Tree

```
Procedure Exact_Search
Input : Q
        /* Exact search query */
Output : Result
        /* Result of the query: found or not found */
Begin
    Result := Not_Found
    If current node P is at leaf level
      If P is equal to the query Q
      Return P
    Else
        For i := 1 to number of children of P
            Set $P_i$ to be the $i^{th}$ children of P
            If the bounding box of $P_i$ encloses the query Q
            Then
                Result := Exact_Search
                Return Result
    Return
End
```

checked. We know that the bounding box of B_2, which is the first child node of B_1, encloses the query. The searching will continue and all child nodes of B_2 will be checked. Bounding boxes of some child nodes of B_2 do not enclose the query and therefore they will not be accessed. Instead, we find that the bounding box of B_7, which is a child node of B_2, encloses the query. Therefore, the searching process proceed to access the node B_7. Since B_7 has an entry to the query Q, the query is found and the searching process is terminated. In this example, three nodes have been accessed, while five nodes should be accessed if the query is searched on the original R-Tree shown in Figure 3.1. Hence, there is an improvement.

If another query S is specified, then the searching will be started again at B_1. As the bounding box of B_1 encloses the query, the searching will continue and all child nodes of B_1 will be checked. We know that the bounding box of B_2 encloses the query and therefore searching will continue and all child nodes of B_2 will be checked. However, bounding boxes of all child nodes of B_2 do not enclose the query. Since we cannot proceed further and no backtracking is allowed, the searching process is terminated. In fact, the query S does not exist in the tree, and the searching process can be terminated because of the property of Redundant R-Tree. In this example, two nodes have been accessed to indicate that the query does not exist in the tree, while three nodes should be accessed if the query is searched on the original R-Tree shown in Figure 3.1.

7.2 Nearest Neighbor Search

In this section, pruning heuristics of nearest neighbor search of Redundant R-Tree will be described. A complete nearest neighbor search algorithm will then be given, and an example will also be shown to demonstrate how the algorithm works.

Since Redundant R-Tree is a variant of R-Tree, its nearest neighbor search algorithm should be based on the nearest neighbor search algorithm of R-Tree with some modifications.

First of all, we focus on pruning heuristics used in nearest neighbor search algorithm of Redundant R-Tree. As we have proved that the improved nearest neighbor search algorithm stated in Chapter 4 outperforms the nearest neighbor search algorithm in [6], Heuristic 5 is used in the nearest neighbor search algorithm of Redundant R-Tree. In addition, one more pruning heuristic is used so that a better performance can be achieved.

Let $O = (o_1, o_2, ..., o_m)$ be a query and $P = (p_1, p_2, ..., p_m)$ be the temporary

Algorithm 7.2 Nearest neighbor search algorithm for Redundant R-Tree

```
Procedure NN_Search
Input : NODE
        /* node to be visited */
        $NN\_DIST_{temp}$
        /* distance from temporary nearest neighbor to the query */
Begin
      If current node P is at leaf level
      Then
            For i := 1 to no. of children of current node
                If $DIST_P < NN\_DIST$
                  Set current node to be nearest neighbor
                  Update $NN\_DIST$
      Else
          Generate Active Branch List of current node

          Calculate $MINDIST$

          Sort the Active Branch List by ascending ordering of $MINDIST$

          For i := 1 to no. of entries in the Active Branch List
              Apply Heuristic 5 and 6 to do pruning

              Call $NN\_Search$

End
```

nearest neighbor of an query O. Let $M = (L, U)$ be the current bounding box where $L = (l_1, l_2, ..., l_m)$ is the lower bounds and $U = (u_1, u_2, ..., u_m)$ is the upper bounds of the bounding box respectively. The additional heuristic is presented as follows:

Heuristic 6 *If the conditions $NN\_DIST_{temp} \leq \min(|o_i - l_i|, \forall i = [1, 2, ..., m])$ and $NN\_DIST_{temp} \leq \min(|o_i - u_i|, \forall i = [1, 2, ..., m])$ are satisfied, then the nearest neighbor can be obtained after all candidate child nodes of the current node have been accessed.*

Theorem 7 *Heuristic 6 is correct.*

Proof: If the conditions $NN\_DIST_{temp} \leq \min(|o_i - l_i|, \forall i = [1, 2, ..., m])$ and $NN\_DIST_{temp} \leq \min(|o_i - u_i|, \forall i = [1, 2, ..., m])$ are satisfied, by Theorem 6, the nearest neighbor must be located within the bounding box of current node M, that is, bounding box of M encloses the nearest neighbor. Property 3 states that the bounding box of A encloses the bounding box of B if and only if node A has an entry to node B. Therefore, the nearest neighbor must be obtained after all candidate child nodes of node M which are entries in the Active Branch List of node M have been accessed. □

Based on the improved nearest neighbor search algorithm and Heuristic 6, the nearest neighbor search algorithm on Redundant R-Tree is obtained and it is given in Algorithm 7.2.

Consider the nearest neighbor search example given in Chapter 5 to see how the nearest neighbor search algorithm works. Assume P is the nearest neighbor of a given query. The searching started at the root, and after certain steps the subtree of node A has been accessed. a_i denotes the temporary nearest neighbor after the subtree of node A has been accessed. Since P is located within the bounding box of node A, by Theorem 6, a_i is P. If the query is not located very near to the boundary of bounding box of node A, that is, the conditions $NN\_DIST_{temp} \leq \min(|p_i - l_i|, \forall i = [1, 2, ..., m]$ and $NN\_DIST_{temp} \leq \min(|p_i - u_i|, \forall i = [1, 2, ..., m]$ are satisfied, then, by Heuristic 6, the searching process will terminate after all candidate child nodes of node A have been accessed. Therefore, the subtree of node B and C will not be accessed even if $NN\_DIST_{a_i} > MINDIST_B$ or $NN\_DIST_{a_i} > MINDIST_C$. In the nearest neighbor search algorithm of R-Tree, node B and node C may also be accessed. It is because the nearest neighbor P may be a child node of either node A, B or C. Furthermore, $MINDIST_A$, $MINDIST_B$ and $MINDIST_C$ are all equal to zero which implies that it is possible to exist an object P' such that it is a child node of either node A, B or C, and $DIST_{P'} \leq DIST_P$.

As the correctness of theorems and pruning heuristics have been proved, the nearest neighbor search algorithm is shown to be correct and it will not miss the nearest neighbor. For the case described above, nearest neighbor search of redundant R-Tree is more efficient than that of R-Tree as subtrees of node B and C do not need to be accessed. When the numbers of data is very large, R-Tree usually has many overlapping regions and the number of objects enclosed by a bounding box is very large too. It implies that the preconditions of Heuristic 6 are very likely to occur. If there is no overlapping region, redundant R-Tree is just a original R-Tree, and hence the performance will be the same in this case.

7.3 Avoidance of Multiple Accesses

We assume that all queries will be executed sequentially, i.e. there is no concurrent querying. Since a node may have more than one *parent* node, it is necessary to eliminate the possibility of multiple accesses of a node in the same search query. In order to do that, a global logical clock in the system should be given. The clock will only be updated once when a new search query is specified and stay unchanged throughout the same query. During the searching process, when a node is being accessed, its timestamp will be checked first and then a new timestamp will be given to the node. The timestamp is used to check whether it has been accessed on this searching transaction. If the timestamp of the node is the same as the value of current logical clock, it can be ensured that the node have been accessed in this transaction, and there is no need to access it again. Hence, the node will be ignored. The chance of multiple accesses on the same node is eliminated.

Chapter 8

Experiment

In this Chapter, a series of experiments will be performed and their results will also be presented. These experiments are used to compare the performance of searching in different content-based index structures. Both exact and nearest neighbor search will be performed on R-Tree, R*-Tree, Redundant R-Tree and Redundant R*-Tree.

8.1 Experimental Setup

We implemented both R-Trees, R*-Tree, Redundant R-Tree and Redundant R*-Tree in C under UNIX on a Sun Ultra-Sparc computer, and used our implementation in a series of performance tests to verify the practicality of the structure and to evaluate the efficiency during searching by both algorithms. Both exact and nearest neighbor search will be used in these experiments.

Clustered, uniform and real data will be used in the experiments. The size of all data set varies from 10000 to 40000. The set of clustered data has 100 clusters and its dimension varies from 4 to 32. The dimensions of the uniform data vary from 4 to 32, and the dimensions of real data vary from 2 to 16.

8.2 Exact Search

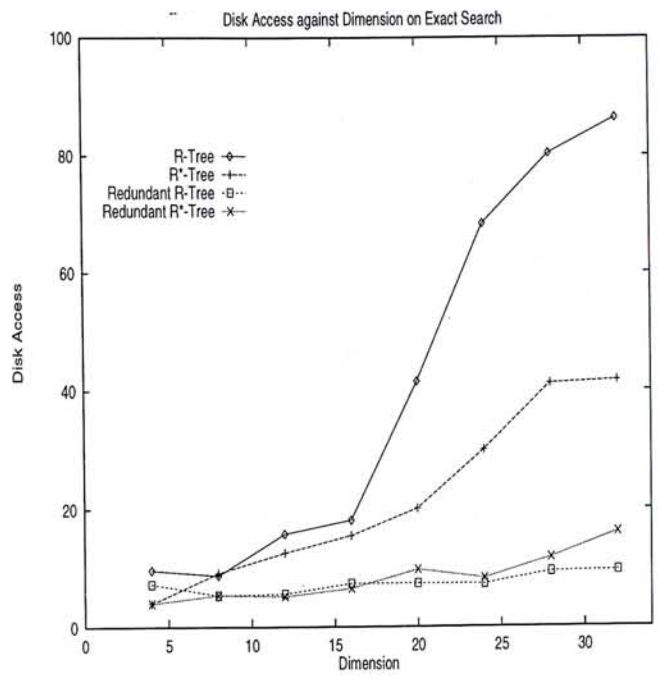


Figure 8.1: Exact search on 10000 clustered data

8.2.1 Clustered Data

In the first set of experiment, one hundred exact search query are performed on the clustered data. The experiments are used to compare the performance of R-Tree, R*-Tree, Redundant R-Tree and Redundant R*-Tree on exact search. Results of this set of experiments are shown in Figures 8.1 to 8.2.

First of all, the performance of the index structures on exact search are evaluated by average disk accesses per exact search query. Figure 8.1 shows the average disk accesses per query against the numbers of dimensions of data. Both R-Tree and R*-Tree perform badly when the numbers of dimensions of data increase. It is because when they are dealing with high dimension data, they have a lot of overlapping nodes and the performance will be degraded rapidly. The

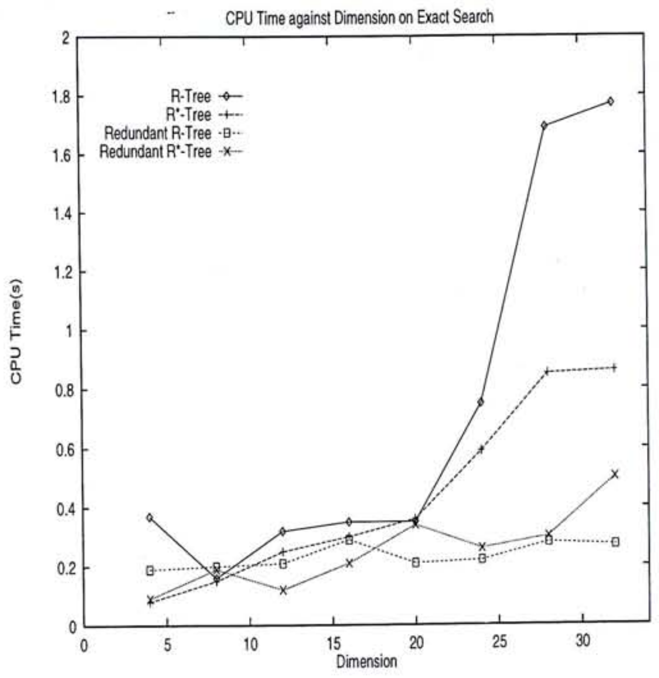


Figure 8.2: Exact search on 10000 clustered data

curves resulted by the increasing rate of bad performances. As what have been described in Chapter 5, there are many backtrackings in exact search on both R-Tree and R*-Tree that cause many unnecessary node access. Furthermore, R-Tree performs worse than R*-Tree because R*-Tree tries to avoid overlapping nodes which can improve its own performance. On the other hand, Redundant R-Tree and R*-Tree outperform R-Tree and R*-Tree because unique search path can be provide for each query.

Figure 8.2 shows the CPU Time used against the numbers of dimensions of data. Similar to the results above, Redundant trees used less CPU Time for searching. However, as larger nodes take more time for processing, the gains of CPU Time saved in Redundant trees are not as many as the gains of the number of disk accesses saved.

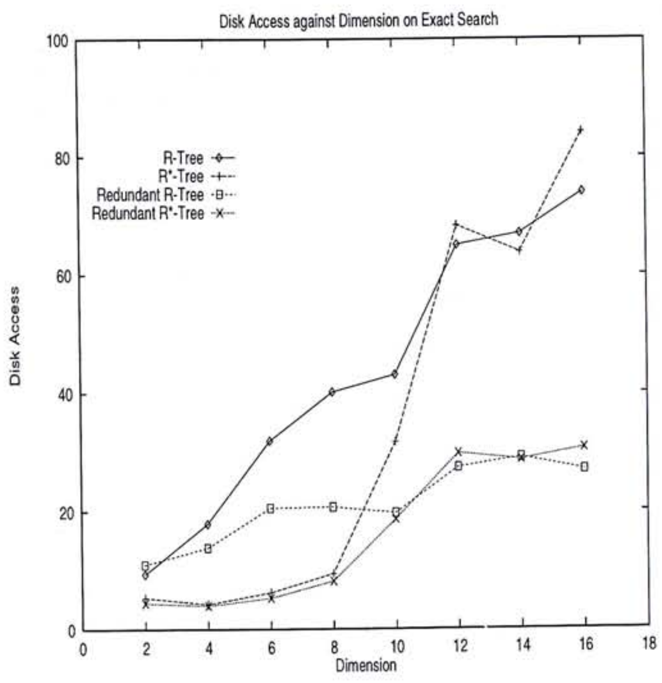


Figure 8.3: Exact search on 10000 real data

8.2.2 Real Data

In this set of experiment, one hundred exact search query are performed on the real data. The experiments are used to compare the performance of R-Tree, R*-Tree, Redundant R-Tree and Redundant R*-Tree on exact search. Results of this set of experiments are shown in Figures 8.3 to 8.4.

First of all, the performance of the index structures on exact search is evaluated by average disk accesses per exact search query. Figure 8.3 shows the average disk accesses per query against the numbers of dimensions of data. R*-Tree is more capable of avoiding overlapping nodes, so its performance is better than R-Tree especially when low dimensional data are being handled. However, when the numbers of dimensions of data increase, R*-Tree cannot effectively reduce overlapping nodes and its performance is almost the same as that of R-Tree. Since R-Tree has many overlapping nodes, the Redundant R-Tree performs

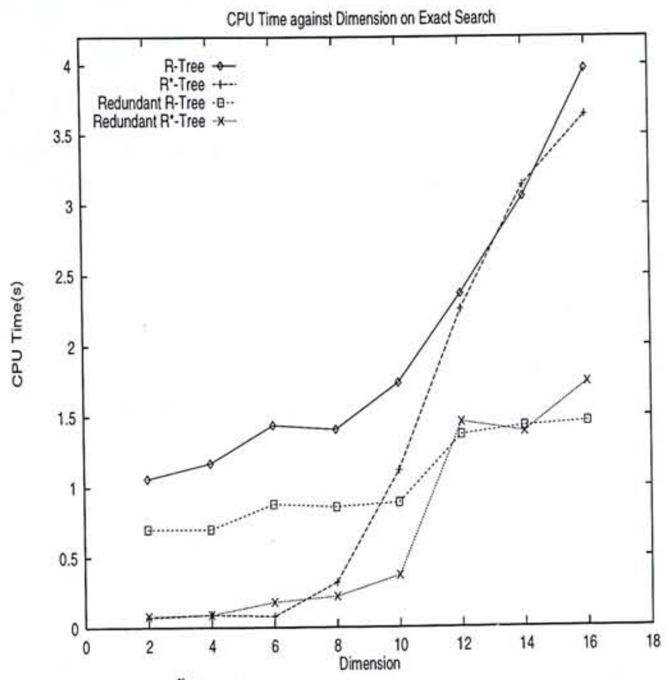


Figure 8.4: Exact search on 10000 real data

slightly worse than R*-Tree. However, the numbers of disk accesses of redundant trees increases steadily that make Redundant R-Tree performs better than R*-Tree when high dimensional data are being handled. Redundant R*-Tree, on the other hand, gives the best performance as few overlapping nodes when low dimensional data are being handled and the steady growth of the numbers of disk accesses that makes it perform better than Redundant R-Tree when high dimension data are being handled.

Figure 8.4 shows the CPU Time used against the numbers of dimensions of data. Similar to the results above, R-Tree performs the worst while Redundant R*-Tree gives the best performance on average. R*-Tree gives better performance than Redundant R-Tree when low dimensional data are used, but Redundant R-Tree performs better than R*-Tree when high dimensional data are used. Since larger nodes induce more overhead for processing, the gains of CPU Time saved in Redundant trees are not as many as the gains of the number of disk accesses saved.

8.3 Nearest Neighbor Search

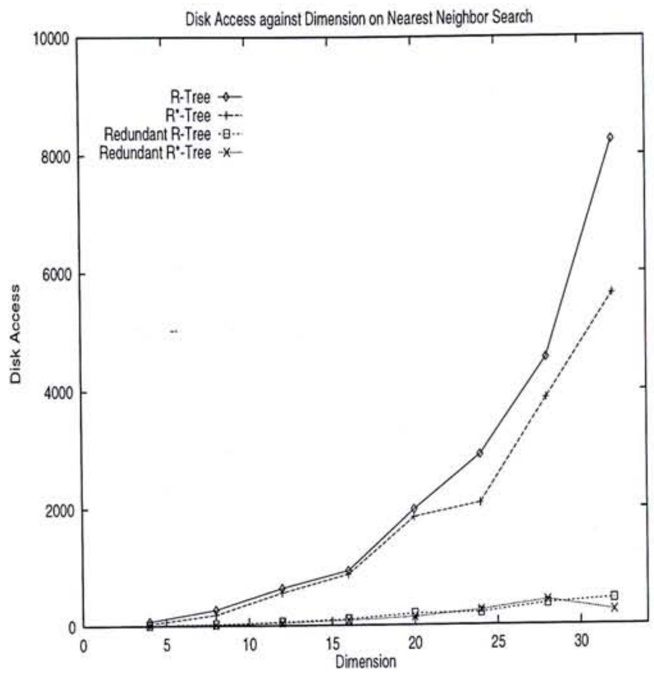
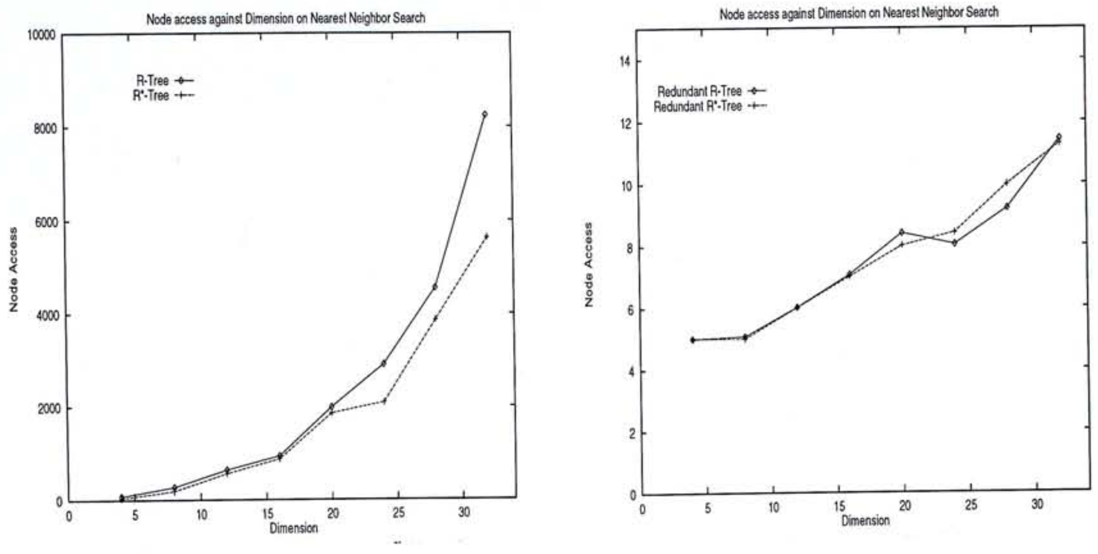


Figure 8.5: Nearest neighbor search on 40000 clustered data

In this section, the performance of nearest neighbor search on different index structures will be tested. The most nearest neighbor will be searched for each query. Nearest neighbor search algorithms described in Chapters 4 and 7 will be used to search on original trees and Redundant Trees respectively. Several metrics will be used to evaluate the performance. Those metrics are the numbers of node accesses per query, the numbers of disk accesses per query, and the percentage of node accesses per query.

8.3.1 Clustered Data

In the first set of experiment, one hundred nearest neighbor search query are performed on the clustered data and each query is need to find the most nearest

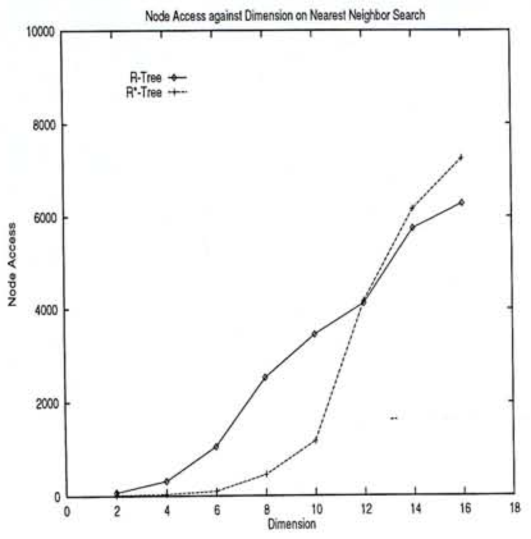


(a) R-Tree and R*-Tree (b) Redundant Trees

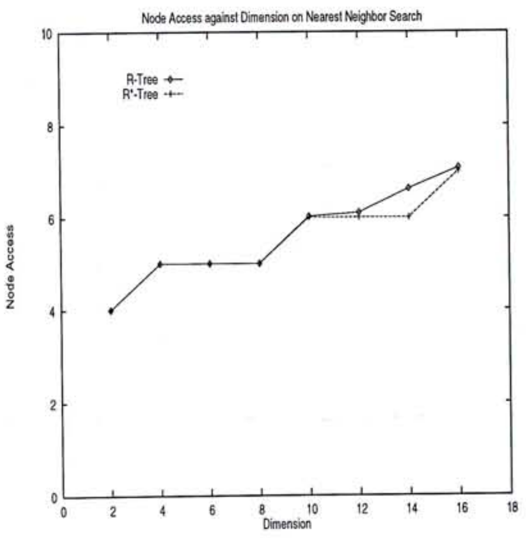
Figure 8.6: Node access against dimension on 40000 clustered data

neighbor. The experiments are used to compare the performance of R-Tree, R*-Tree, Redundant R-Tree and Redundant R*-Tree on nearest neighbor search. Results of this set of experiment is shown in Figures 8.5 to 8.7.

First of all, the performance of the index structures on nearest neighbor search is evaluated by average disk accesses per nearest neighbor search query. Figure 8.5 shows the average disk accesses per query against the numbers of dimensions of data. Both R-Tree and R*-Tree perform badly when the numbers of dimensions of data increase. It is because when they are dealing with high dimensional data, they have a lot of overlapping nodes and the performance will be degraded rapidly. The curves resulted by the increasing rate of bad performances. As what have been described in Chapter 5, there are a lot of backtrackings in nearest neighbor search on both R-Tree and Redundant R*-Tree that cause many unnecessary node access. On the other hand, Redundant R-Tree and R*-Tree outperform R-Tree and R*-Tree because the searching is much more



(a) R-Tree and R*-Tree



(b) Redundant Trees

Figure 8.7: Percentage of node access against dimension on 40000 clustered data

deterministic by making use of redundancy on index trees.

Average node access per query is shown in Figure 8.6. Similar to the result measured by average disk accesses per query, both R-Tree and R*-Tree perform badly when the numbers of dimensions of data increase, while Redundant R-Tree and Redundant R*-Tree outperform R-Tree and R*-Tree. The difference of node accesses between Redundant trees and original trees is larger than the difference of disk accesses. It is because each node in R-Tree and R*-Tree is fitted into one page size, while the size of each node in Redundant trees may be larger than one page. The numbers of node accesses are proportional to the numbers of backtrackings during searching. The result shows that the numbers of node accesses of Redundant trees are closed to the height of trees which show that fewer backtracking occurred. The percentage of node accesses per query are shown in Figure 8.7. Similar to the result in measuring the number of node accesses, searching on Redundant trees needs fewer percentage of node accesses

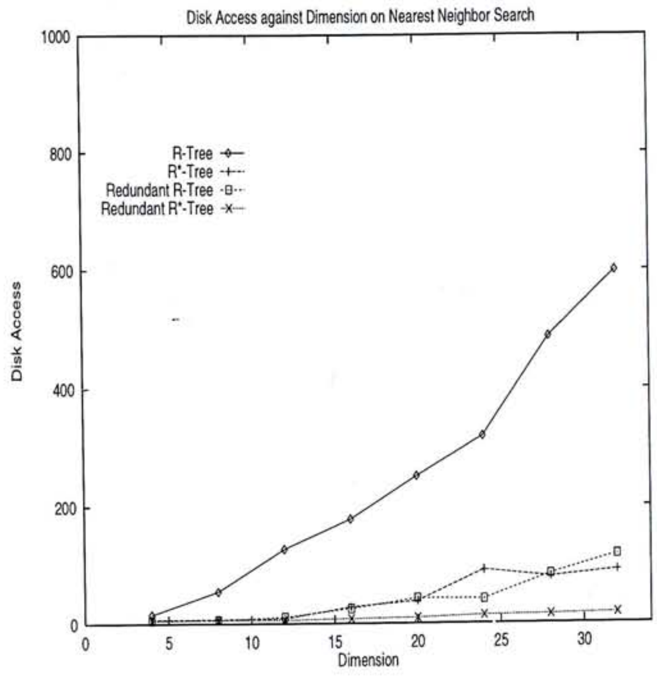


Figure 8.8: Nearest neighbor search on 40000 uniform data

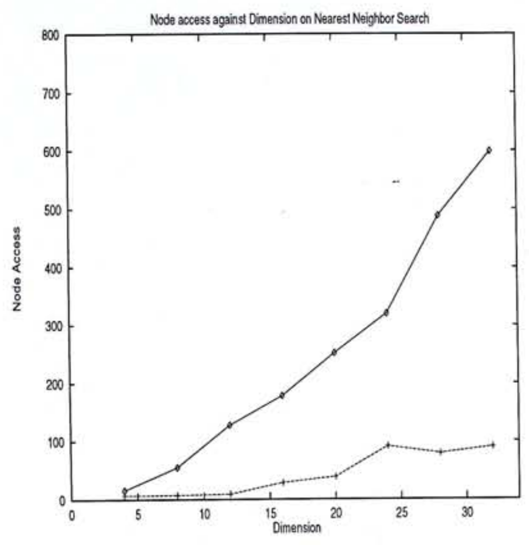
on average.

8.3.2 Uniform Data

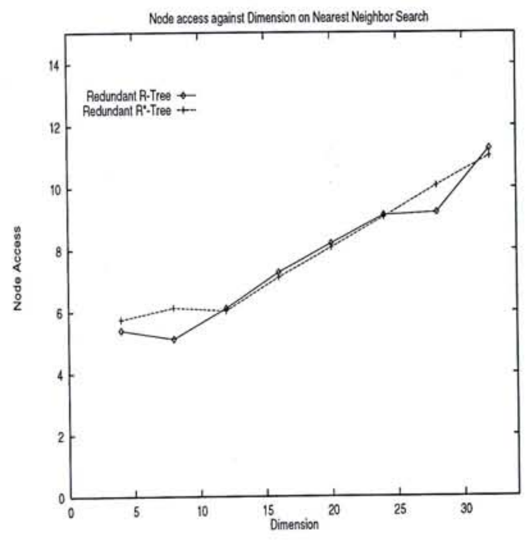
In the second set of experiment, 100 Nearest neighbor query are performed on the uniform data. Results of this set of experiment is shown in Figures 8.8 to 8.10.

Once again, the performance of our proposed Redundant R-Tree and Redundant R*-Tree outperform R-Tree and R*-tree in this set of experiment. However, since the uniform data have less overlapping, the performance of R-Tree and R*-Tree is slightly better than that of dealing with clustered data. Therefore, the speedup factor of Redundant R-Tree and R*-Tree is slightly smaller.

Figure 8.8 shows the average disk accesses per query against the numbers of dimensions of data. Both R-Tree and R*-Tree perform badly when number of



(a) R-Tree and R*-Tree

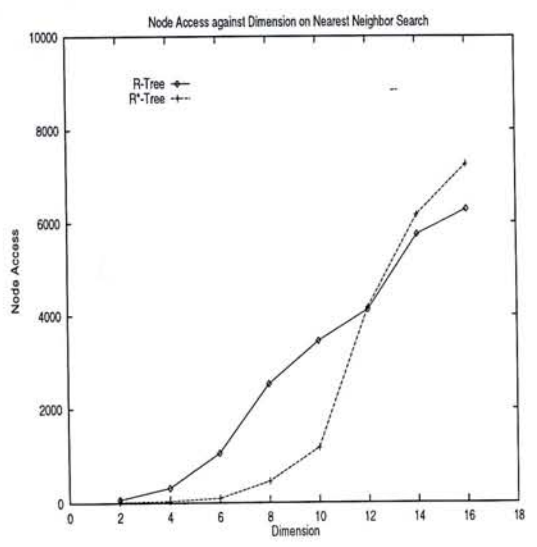


(b) Redundant Trees

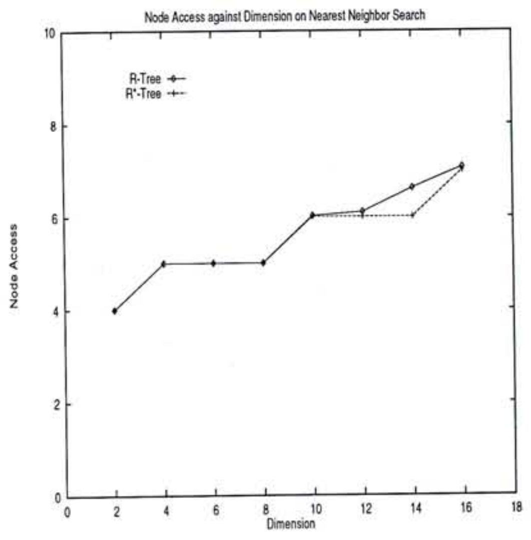
Figure 8.9: Node access against dimension on 40000 uniform data

dimension of data increased. However, R*-Tree is shown to perform better than R-Tree. It is due to the fact that overlapping in uniform data is less that that of clustered data. Since R*-Tree is more capable of avoiding overlapping nodes, it significantly outperforms R-Tree. As R-Tree performs badly, the performance of Redundant R-Tree is similar to that of R*-Tree because too much overlapping nodes in R-Tree cause larger node size in Redundant R-Tree. On the contrary, Redundant R*-Tree gives the best performance since R*-Tree has better performance than R-Tree.

Average node accesses per query is shown in Figure 8.9. Similar to the result measured by average disk accesses per query, R-Tree gives the worst performance. R*-Tree performs better. Redundant R-Tree and Redundant R*-Tree access fewer nodes on average. As the size of node in Redundant trees is larger or equal to the size of node of original trees, Redundant R-Tree accesses fewer



(a) R-Tree and R*-Tree



(b) Redundant Trees

Figure 8.10: Percentage of node access against dimension on 40000 uniform data

nodes on average than R-Tree and R*-Tree. As similar as before, Redundant R*-Tree accesses the fewest number of nodes. The percentage of node accesses per query are shown in Figure 8.10. Similar to the result in measuring the number of node accesses, searching on Redundant R*-Tree needs the fewest percentage of node accesses.

8.3.3 Real Data

Experiments have also been done on real data. Figures 8.11 and 8.13 show the results.

In the real data, there are a lot of overlapping and it makes the performance of R-Tree and R*-Tree become worse. When it is compared to those results of using uniform and clustered data, the results of using real data show R-Tree and R*-Tree give worse performance.

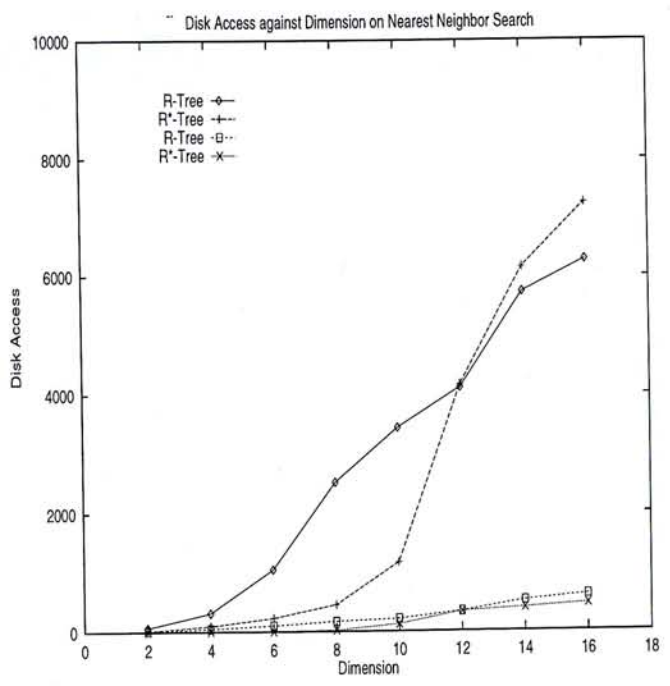
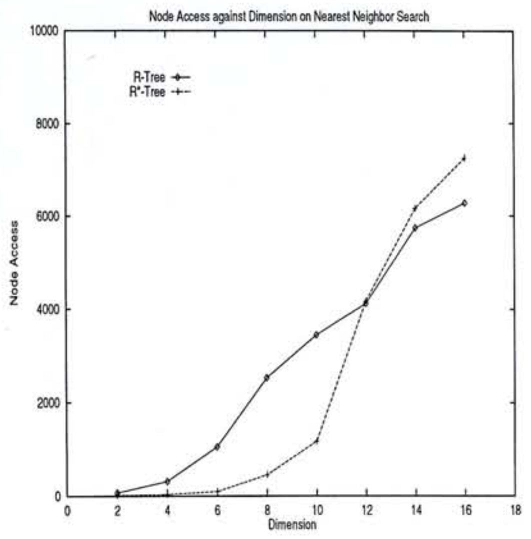


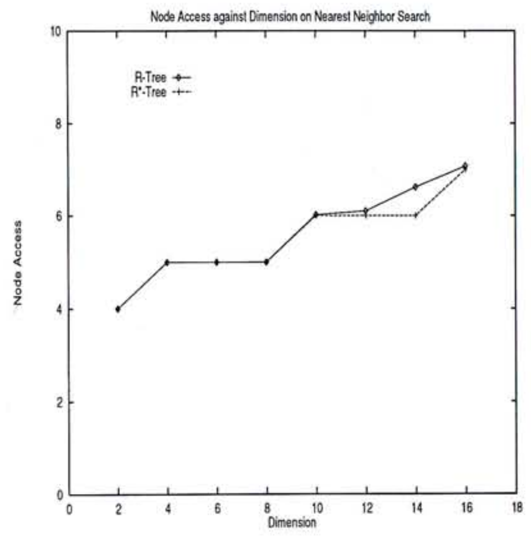
Figure 8.11: Nearest neighbor search on 40000 real data

Figure 8.11 shows the average disk accesses per query against the numbers of dimensions of data. Both R-Tree and R*-Tree perform badly when the numbers of dimensions of data increase. Initially, R*-Tree is shown to perform better than R-Tree, since R*-Tree is more capable of avoiding overlapping nodes. As the dimension of data increased, more overlapping nodes are created which make both R-Tree R*-Tree perform badly and their performance are almost the same. On the contrary, Redundant R-Tree and R*-Tree perform better than original trees.

Average node accesses per query is shown in Figure 8.12. Similar to the result measured by average disk accesses per query, R-Tree gives the worst performance. R*-Tree performs better when the dimension of data is low, and their performance are almost the same when high dimensional data are being dealt with. Redundant R-Tree and Redundant R*-Tree access fewer nodes on average. The percentage of node accesses per query are shown in Figure 8.13. Similar



(a) R-Tree and R*-Tree



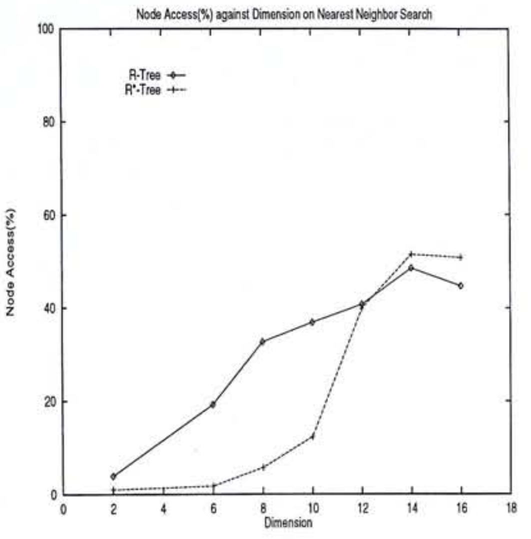
(b) Redundant Trees

Figure 8.12: Node access against dimension on 40000 real data

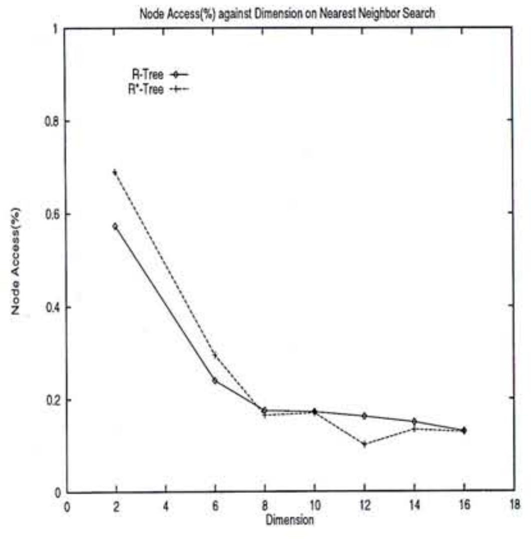
to the result in measuring the numbers of node accesses, searching on Redundant R-Tree and R*-Tree have fewer percentage of node accesses. There is a difference that the percentage of node accesses of the Redundant trees tend to decrease with the numbers of dimensions of data. It is because high dimensional real data induce a lot of overlapping nodes. The presence of overlapping nodes make Redundant Trees become larger in size. The increase of their size are faster than the increase of the height of trees which is proportional to the number of node accesses per search query. Therefore, those curves for redundant trees have negative slopes.

8.4 Discussion

The results show that Redundant R-Tree and R*-Tree outperform the original R-Tree and R*-Tree on both exact and nearest neighbor search. However, it should be noted that the performance of redundant tree can be affected by the



(a) R-Tree and R*-Tree



(b) Redundant Trees

Figure 8.13: Percentage of node access against dimension on 40000 real data

degree of overlapping of their nodes. As we have mentioned before, too much overlapping nodes in trees make larger nodes size in Redundant trees which induces overheads on both processing and the numbers of page accesses for each search query. Furthermore, too much overlapping nodes would also increase the size of the trees. Figures 8.14 and 8.15 show the ratio of storage against the numbers of dimensions of uniform and real data respectively. In order to optimize their performance, an effective methods for reducing overlapping should be used.

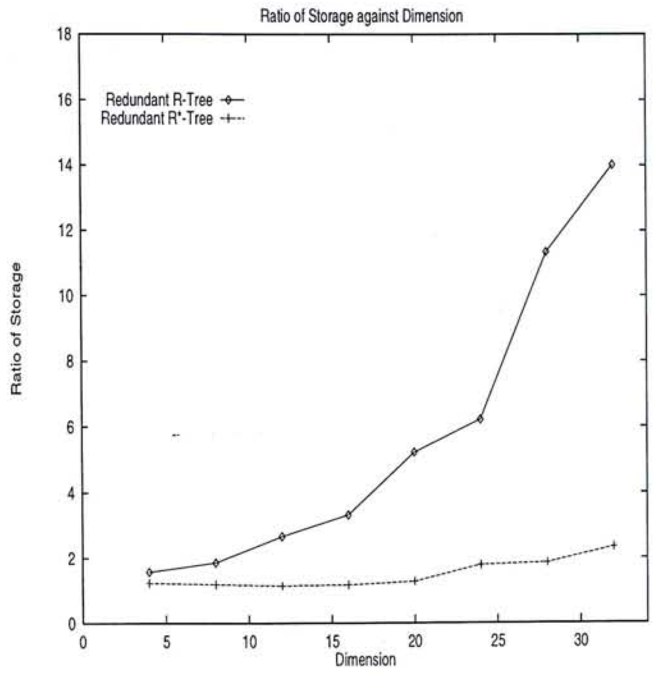


Figure 8.14: Ratio of storage on 40000 uniform data

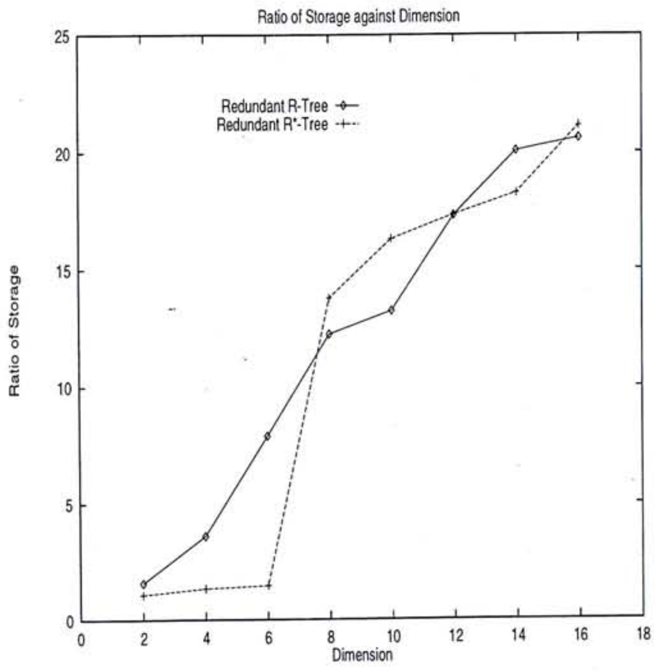


Figure 8.15: Ratio of storage on 40000 real data

Chapter 9

Conclusions and Future Research

9.1 Conclusions

Several content-based index structures such as R-Tree, R^+-Tree and R^*-Tree, and their searching methods have been briefly introduced. The cause of the inefficiency of nearest neighbor search in R-Tree have also been presented. In order to improve the performance, a new pruning heuristic for nearest neighbor search have been derived. It is shown that the new pruning heuristic can replace those old pruning heuristics, while the replacement will not increase the number of node access of any nearest neighbor search query, which means that the performance will not be degraded. Since the new heuristic does not make use of a metric *MINMAXDIST* which is very computational expensive, the performance is improved. Based on the new pruning heuristic, an improved nearest neighbor search algorithm as well as an improved N-nearest neighbor search algorithm for R-Tree and its variants have been designed. A series of experiments have been carried out, and the results show that the new nearest neighbor search algorithm uses less CPU time for processing, while it does not increase number of node access. Therefore, the improved nearest neighbor search algorithm outperforms the original one.

Besides, we found that the general performance of searching on R-Tree and its variants are not satisfactory. Overlapping nodes of R-Tree have also been described. The presence of overlapping nodes induces many backtrackings during searching, and it therefore degrades the performance on data retrieval. We try to improve it by adding redundancy to the tree. The idea of adding redundancy on R-Tree have been introduced. We have presented the motivation of the design of our Redundant Tree, which is a variant of R-Tree with redundancy, and we have examined algorithms to build the Redundant Tree. Also, exact and nearest neighbor search algorithms of Redundant Tree have been given. Based on the properties of Redundant Tree, our proposed algorithms try to reduce the number of node access during searching. We have performed experiments to compare these algorithms with the original algorithms. The proposed searching algorithms in Redundant Tree perform better than those of R-Tree and R*-Tree. We conclude that Redundant Tree outperforms R-Tree and R*-Tree.

9.2 Future Research

Future research could examine methods to use less storage to build a Redundant R-Tree. Also, we should minimize the sorting overhead in the nearest neighbor search. In fact, the problems can be solved by reducing but not strictly eliminating overlapping nodes in the index tree. Therefore, a better algorithm for splitting should be designed so that a moderate degree of overlapping is allowed. R-Tree, which does not eliminate any overlapping node on purpose, shows its inefficiency on searching because of the presence of overlapping node. R$^+$-Tree, which does eliminate all overlapping nodes on purpose, shows its inefficiency on searching because there are many nodes with large margins. A mixture of R*-Tree and R$^+$-Tree may be a good choice because R*-Tree tries to keep minimal margin of bounding boxes, while R$^+$-Tree tries to eliminate overlapping nodes.

In fact, keeping margin and overlapping minimum are important to the performance of searching.

X-Tree is a variant of R-Tree and it is designed by Stefan Berchtold *et al*, in [9]. In [9], Berchtold designs a split algorithm which uses split history of a node so that the split will be overlap-minimal. Besides, he introduces supernode to the tree structure. When a split makes two heavy overlapping nodes, a supernode will be produced instead of splitting. We are doing experiments to compare performance between Redundant Tree and X-Tree, and result will be released soon.

A more efficient algorithm for introducing redundancy should be designed. Also, redundancy idea can be extended to other multi-dimensional index structures, for example Vp-Tree.

Bibliography

[1] Rohini K. Srihari, *"Automatic indexing and Content-Based Retrieval of Captioned Images"*, IEEE Computer, volume 28, number 9, pages 49-59, September, 1995

[2] Anne Brink, Sherry Marcus, V.S. Subrahmanian, *"Heterogeneous Multimedia Reasoning"*, IEEE Computer, volume 28, number 9, pages 33-39, September, 1995

[3] Virginia E. Ogle, *"Chabot: Retrieval from a Relational Database of Images"*, IEEE Computer, volume 28, number 9, pages 40-48, September, 1995

[4] Antonin Guttman, *"R-Trees: A Dynamic Index Structure For Spatial Searching"*, Proceedings of ACM SIGMOD Int. Conf. on Mangagement of Data, pages 47-57, 1984

[5] Timos Sellis, Nick Roussopoulos, Christos Faloutsos *"The R+-Trees: A Dynamic Index For Multi-Dimensional Objects"*, Proceedings of the 13th Very Large Database(VLDB) Conference, pages 507-518, 1987

[6] Nick Roussopoulos, Stephen Kelley, Frederic Vincent, *"Nearest Neighbor Queries"*, Proceedings of ACM SIGMOD Int. Conf. on Mangagement of Data, pages 71-79, 1995

[7] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger, *"The R^*-Tree: An Efficient and Robust Access Method for Points and Rectangles"*, Proceedings of ACM SIGMOD Int. Conf. on Mangagement of Data, pages 322-331, 1990

[8] Tzi-cker Chiueh, *"Content-Based Image Indexing"*, Proceedings of the 20th Very Large Database(VLDB) Conference, pages 582-593, 1994

[9] Stefan Berchtold, Daniel A. Keim, Hans-Peter Kriegel, *"The X-Tree: An Index Structure for High-Dimensional Data"*, Proceedings of the 22nd VLDB Conference, 1996

[10] Peter N. Yianilos, *"Data Structures and Algorithms for Nearest Neighbor Search in General Metric spaces"*, Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms, pages 311-321, 1992

[11] King-Ip Lin, H. V. Jagadish, Christos Faloutsos, *"The TV-tree - an Index Structure for High-dimensional Data"*, Very Large Database(VLDB) Journal, volume 3, pages 517-542, 1995

[12] Hanan Samet, *"The Design and Analysis of Spatial Data Structures"*, Addison-Wesley, 1989

[13] Venkat N. Gudivada, Vijay V. Raghavan, *"Content-Based Image Retrieval Systems"*, IEEE Computer, volume 28, number 9, pages 18-22, September, 1995

[14] Niblack W. et al, *"The QBIC Project: Querying Images By Content Using Color, Texture and Shape"*, SPIE 1993 International Sumposium on Electronic Imaging: Science and Technology, Conference1908, Storage and Retrieval for Image and Video Databases, volume 1908, pages 173-187, February 1993

[15] Niblack W. et al, *"The QBIC Project: Query By Images Content Using Multiple Objects and Multiple Features: User Interface Issues"*, Proceedings of the 1st International Conference on Image Processing, volume II, pages 76-80, 1994

[16] Niblack W. et al, *"The QBIC Project: Indexing for Complex Queries on a Query-By-Content Image Database"*, International Conference on Pattern Recognition, volume 1, pages 142-146, 1994

[17] Niblack W. et al, *"Query by Image and Video Content: The QBIC System"*, IEEE Computer, volume 28, number 9, pages 23-32, September, 1995

[18] Dimitris Papadias, Yannis Theodoridis, Timos Sellis, Max J. Egenhofer, *Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees*, Proceedings of ACM SIGMOD Int. Conf. on Mangagement of Data, pages 92-103, 1995

[19] Michael Freeston, *A General Solution of the N-dimensional B-trees*, Proceedings of ACM SIGMOD Int. Conf. on Mangagement of Data, pages 80-91, 1995

[20] S.B. Yao, *Approximating Block Accesses in Database Organizations*, Communications of the ACM, pages 260-261, 1987

[21] Patrick Valduriez, *Join Indices*, ACM Transactions on Database Systems, pages 218-246, 1987

[22] Ibrahim Kamel, Christos Faloutsos *On Packing R-Tree*, Proceedings of Information and Knowledge management(CIKM), pages 218-246, November 1993

[23] Nick Roussopoulos, Daniel Leifker *Direct Spatial Search on Pictorial Databases Using Packed R-Trees*, Proceedings of ACM SIGMOD, pages 17-31, 1985

[24] Yannis Theodoridis, Timos Sellis *A Model for the Prediction of R-Tree Performance*, Proceedings of Database Systems, pages 161-171, 1996