Efficient Spatial Data Structure for Multiversion Management of Engineering Drawings

Yasuaki Nakamura
Department of Computer and Media Technologies,
Hiroshima City University
Hiroshima, 731-3194, Japan

and

Hiroyuki Dekihara
Department of Information Technology,
Hiroshima International University
Kure, 737-0112, Japan

ABSTRACT

In the engineering database system, multiple versions of a design including engineering drawings should be managed efficiently. The paper proposes an extended spatial data structure for efficient management of multiversion engineering drawings. The R-tree is adapted as a basic data structure. The efficient mechanism to manage the difference between drawings is introduced to the R-tree to eliminate redundant duplications and to reduce the amount of storage required for the data structure. The extended data structures of the R-tree, *MVR* and *MVR** trees, are developed and the performances of these trees are evaluated. A series of simulation tests shows that, compared with the basic R-tree, the amounts of storage required for the *MVR* and *MVR** trees are reduced to 50% and 30%, respectively. The search efficiencies of the R, *MVR*, and *MVR** trees are almost the same.

Keywords: Spatial Data Structure, R-tree, Version Management, Design Database, CAD.

1. INTRODUCTION

A computer aided design system (CAD) is the collection of software for creating or synthesizing a design, analyzing it for design correctness, managing the storage and organization of the design data, and managing the process of design flow, that is, the controlled sequencing of design applications to yield a correctly designed artifact. The engineering design data usually consist of a set of documents and drawings or diagrams. In CAD design management, a design database system deals with the storage and retrieval of design data and its consistent update. When the update process creates a new version of design from an old version, the old version as well as the new one must be kept in the database because they may be referenced from other designs. Therefore, the management of updating processes and versions of design is very important in the design database.

Several version modeling techniques for the engineering database [1], and multi-version management structures for text data or non-spatial data [2-5] have been also proposed. However, the data structures for version management of drawings have not been proposed in the version modeling. Usually, a drawing contains a large number of spatial objects, such as points, line segments, shapes, and so on. The update process adds and deletes objects in a drawing. The drawing is then saved as a new version. Spatial data structures such as the R-tree [6] and the MD-tree [7] have been used in the CAD database to manage the drawings and spatial objects efficiently. As for the management of spatial temporal objects, several data structures [8-10] for version management of spatial objects and for moving objects have been proposed. In these data structures, time stream is unique, that is, each object is managed based on the absolute time and position. In design databases, multiple versions are generated from a version. This can be considered multiple time streams exist. Data structures managing multiple versions of drawings have not been proposed. In the paper, the version management model of the engineering drawings is proposed. Our data structure, called the *MVR*-tree, is an extension of the R-tree that can manage the multiple versions of drawings. After evaluating the *MVR*-tree, we also propose an improved data structure, called the *MVR**-tree.

In the following sections, a version model of engineering drawings and a spatial data structure, called the R-tree, are described in Section 2. In section 3, our data structures, the *MVR* and *MVR** trees, are presented. A series of experimental results is shown in 4. As a result, it is shown that the *MVR**-tree has much better performances than the *MVR* and the simple R-tree based method.

2. SPATIAL DATA STRUCTURE AND VERSION MANAGEMENT

Suppose design data X and Y have their own version histories and a version of X references some version of design data Y. Fig.1 shows an example version histories and references. Version 1 of Y, Y.1, is referenced from versions 1 and 2 of X; X.1 and X.2. Y.2 and Y.3 are referenced from X.3 and X.3.1 respectively. Each version of design data has a drawing, for example, a draft of a part or circuit diagram. A new version is created by modifying a portion of the drawing. In the design database, a hierarchical spatial data structure is usually used to manage the spatial objects in drawings. Especially, the R-tree [6] is one of the most popular data structure for spatial data management. In the section, at first, the algorithm of the R-tree is described briefly, because our proposed data structure is developed by extending the R-tree.

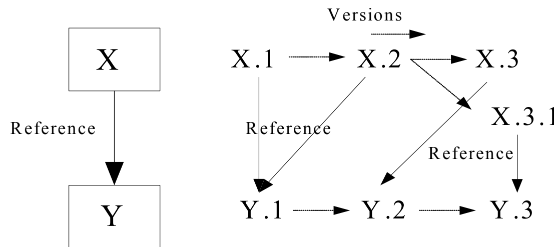


Fig. 1 Versions and references of the engineering design

2.1 The R-tree [6]

Fig.2 shows an example R-tree. In the R-tree, each node corresponds to a rectangular region that encloses all regions of the lower nodes. The root corresponds to a region enclosing all data objects in the R-tree. Objects are only stored in leaf nodes. An internal node has at most **M** child nodes, and a leaf has at most **M'** data. Both an internal node and a leaf manage a rectangular region, called Minimum Bounding Box (**MBB**), that encloses regions below that node and objects in the leaf, respectively. **M** and **M'** in an R-tree in Fig2 are set to 3. When an object is inserted into full leaf L, a new leaf is created and at least (C x **M'**) objects in L are moved to the new leaf, where C is a constant. Usually, C is set to 0.4, because 0.4 gives better performances. The new leaf is added as a child of the L's parent

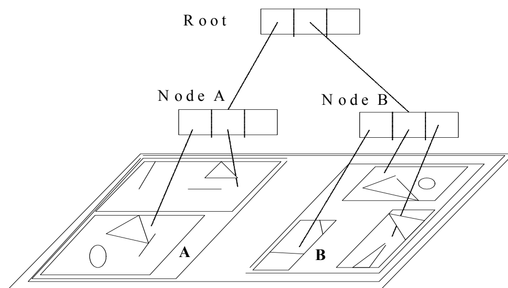


Fig.2 An example R-tree

node. If the number of children of a node exceeds **M**, a new node is created and at least (C x **M**) children of the node are moved to the new node. The details of node splitting algorithm is described in [6]. In the following examples, we assume **M** and **M'** are set to be 3 and C is equal to 2/3. Therefore nodes and leaves contain at least 2 child nodes and objects.

2.2 Version Management of Engineering Drawings

A simple method to manage the multiple versions of a drawing is to create an R-tree corresponding to each version of the drawing, and to manage these R-trees in the database, as shown in Fig.3. The amount of storage required for these R-trees is proportional to the number of versions, when the number of objects in each version is the same. Generally, an update process modifies a small number of objects in a version or a small portion of the drawing in general, and a large part of the drawings and R-trees between versions are unchanged. Therefore, sharing the unchanged part between versions, the amount of the storage required can be reduced without the loss of search performances. Based on this idea, we expand the R-tree to share the unchanged objects and nodes with other R-trees.

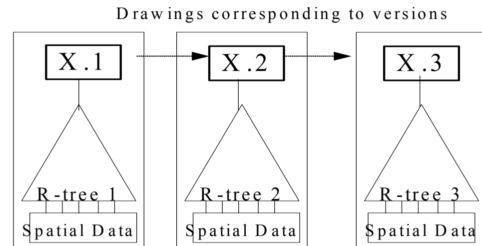


Fig.3 Management of drawings by R-trees corresponding to versions of a design.

3. DATA STRUCTURE FOR VERSION MANAGEMENT: THE MVR-TREE

3.1 The *MVR*-tree

In a simple version management by R-trees, to create a new version of a design data, we create a entire copy of an old version, and then modify and save it as a new version. Even if we modify a small part of the design, the entire design data including unchanged data in the version are saved in the new version. To reduce the amount of storage, we propose a new data structure, called the *MVR-tree*, in which the unchanged objects and nodes are shared between several R-trees corresponding to versions. In the *MVR-tree*, we introduce the path copy method to manage the multiple version R-trees. The path copy method is described with referring Fig.4. Each node and leaf of the trees in Fig.4 can have at most 3 child nodes and 3 objects.

Assume new version **V2** is created from version **V1** of an R-tree. First, new root **R2** corresponding to **V2** is created by copying **R1**. **R1** and **R2** have the same child nodes as shown in Fig.4 (a). When object **O1** is inserted into leaf **L1** through node **N1**, all the nodes on the path from **R2** to **L1** are copied to the tree rooted with **R2**; **N1** and **L1** are copied as **N1'** and **L1'** in **V2**. **O1** is then inserted in **L1'**, if **L1'** is not full. The occupancy of a leaf is indicated by gray color in a box. For examples, leaf **L2** is

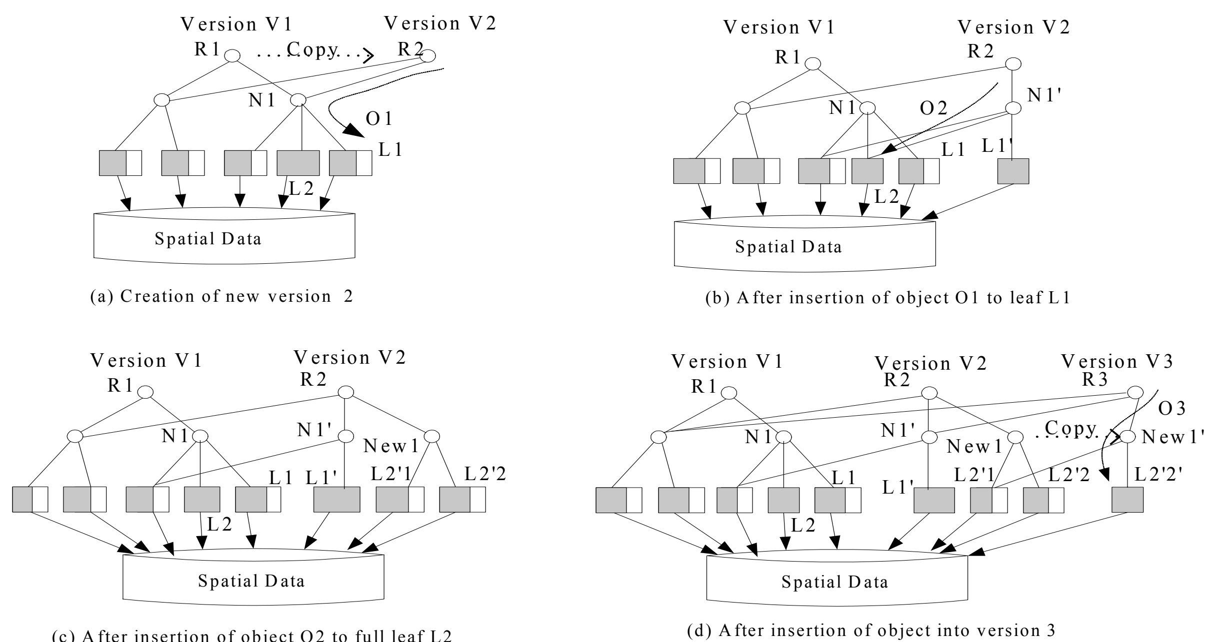


Fig.4 The MVR-tree managing versions

full, but others are not. An *MVR-tree* in Fig.4 (b) is obtained after inserting **O1**. When object **O2** is inserted into full leaf **L2** through **R2**, a copy of **L2**, **L2'1**, is created. New leaf **L2'2** is then created and (C x **M'**) objects in **L2'1** are moved to **L2'2**. **O2** is inserted into either **L2'1** or **L2'2**. As a result, **N1'** has four child nodes. Since any node cannot have more than 3 child nodes, new node **New1** is created to manage **L2'1** and L**2'2**. Then, **New1** is added to **R2** as a child. Fig.4 (c) illustrates this case.

If a new version is created from the **k**-th version tree, a new root is created by copying the root of the **k**-th version. In Fig.4 (d), the third version, **V3**, is created from the second version tree **V2**, and an object, **O3**, is inserted in **L2'2**. Eventually, node **New1** and leaf **L2'2** are copied to **New1'** and **L2'2'**, respectively. Object **O3** is inserted into **L2'2'**. The actual algorithms of creation of a new version and insertion of an object are presented in the followings.

(a) Construct a new version

CreateNewVersionFrom(V_k)
V_k: version of the R-tree.
/* create a new version by modifying version V_k. */
S1: Create new root R_{k+1} corresponding to a new version.
S2: Set the child pointers of V_k's root to R_{k+1}. Return R_{k+1}.

(b) Insert an object

<u>Insert</u>(V_n, O_x)
V_n: R-tree version
O_x : Object
/* insert object O_x into version V_n */

S1: Let the root of V_n be R.
S2: Tracing from R to lower nodes, find leaf L in which O_x should be inserted.
S3: Copy the nodes and leaf L on the path from R to L that are not in V_n into V_n.
S4: Insert O_x into copied leaf L1 of L, if L1 is not full. Otherwise, create new leaf L2 and move (c x **M'**) objects in L1 to L2. Insert O_x into L1 or L2.
S5: If a new leaf is created, adjust the R-tree. This procedure is the same as that of the R-tree.

(c) Delete an object

<u>Delete</u>(V_n, O_x)
V_n: R-tree version
O_x : Object
/* delete object O_x from V_n. */
S1: Let the root of V_n be R.
S2: Tracing from R, find leaf L in which O_x exists.
S3: If L is in V_n and is not shared by other versions, remove O_x from L. If L satisfies the underflow condition, remove L and re-insert the objects in L. (This is the same algorithm of the R-tree.) Return.
S4: Copy the nodes and leaf L on the path from R to L if they are not in V_n into V_n.
S5: Let L' be a copy of leaf L. Remove O_x from copied leaf L'.
S6: If L' satisfies the underflow condition, remove L' from V_n and adjust the tree by re-inserting the objects in L'. (This is the same algorithm of the R-tree.)

(d) Search objects
Spatial searches such as nearest neighbor searches and range searches for a designated version can be performed efficiently both in the simple R-tree based method and the *MVR*-tree. It is well-known that the search performances using the hierarchical

data structures are proportional to the number of accessed nodes during a search process.

In the R-tree, each node and leaf manages the minimum bounding box (**MBB**) that encloses all the objects below that node and in the leaf. The following Range_search procedure can find objects intersecting a give region using the MBB. The search procedures using in the R-tree such as the nearest neighbor searches and range searches can be applicable to the *MVR*-tree.

Range_searchst(V_n, **Range**)
V_n: R-tree version
Range: a search region
/* find objects intersecting a given region, **Range**. */
S1: Let the root of V_n be R.
S2: Trace each child node whose **MBB** intersects **Range** from R until a leaf is found. If the **MBB** of the leaf, L, intersects **Range**, collect objects in L intersecting **Range**.
S3: Return collected objects.

3.2 Features of The MVR-tree

The *MVR*-tree contains the R-trees corresponding to multiple versions of drawings without the duplication of nodes. When a root of an R-tree corresponding to a version in the *MVR*-tree is designated, the structural properties of that R-tree in the *MVR*-tree are the same as the original R-tree managing the objects at that version. Therefore the search performances are expected to be almost equal to the original R-tree, although the MVR-tree has an advantage that the number of nodes and leaves is less than that of the simple R-tree based version management method.

Assume that the modification of objects is applied to a small area and that the number of objects inserted or deleted is small. The *MVR*-tree copies a small number of nodes and leaves and creates a small number of split nodes and leaves. In other words, the unchanged portion between a former version and a new version is shared by both versions. Compared to the simple method in 2.2, the *MVR*-tree has an advantage that the amount of storage required for the *MVR* is much less than that for the simple method. The experiments in section 4 reveal the advantage of the *MVR*-tree to the simple version management of the R-tree.

On the other hand, even though the number of modified objects is small, the number of duplicated nodes and leaves increases if the modification affects the wide area of a drawing. This is because even if one object is inserted in a leaf, the nodes on the path from the root to the leaf must be copied. Therefore, as the modification applies to a wider area, the number of copied nodes and leaves increases considerably even the number of modified data is small. The experimental results showing this aspect are also presented in section 4.

3.3 Improved MVR-tree: The MVR*-tree

The *MVR-tree* is efficient when the modifications are applied to a small area. However, even if the number of modified objects is small, the performances become worse when the modifications are applied to a wide area.

To overcome the shortcoming of the *MVR-tree*, the *MVR**-tree is developed by reducing the duplication of leaves in the *MVR-tree*. Namely, in the *MVR**-tree, when an object is inserted into a leaf with a space, the object is inserted into the leaf without making a copy. In the case, since objects belonging to the different versions exit in a leaf, some token indicating active objects in the versions must be added to the data structure. This idea reduces the number of the duplications of leaves. Since the object occupancy rate in leaves is known to be 60~70% in the R-tree, the small number of modifications would not lead the large number of leaf splits if the modified area is wide.

The idea is illustrated in Fig.5. We add a new structure, called an active object identifier (*AOI*), between a lowest internal node and a leaf. *AOI* consists of an array of Boolean values and a pointer to a leaf. Boolean values indicate which objects in the leaf are active at the version. For examples, *AOI 1* in Fig.5 (a) has value 1 on the first and second elements, and value 0 on the third element. This means that the first and second objects in a leaf are active at version **V0** but the third object is not active in versions **V1** and **V2**. After inserting **O1**, as shown in Fig.5 (b), all elements of *AOI 1'* have value 1. *AOI 1* and *AOI 2* point the same leaf with 3 objects. However, the third object in the leaf is not active in version 1 because the third element of *AOI 1* is 0.

In the actual implementation, *AOI* is integrated into an internal node structure. This modification increases both the amount of storage for the node, and CPU times required for insertion of objects and objects searches a little. However, since the access to the nodes and leaves in the secondary memory takes the major time in the searches and insertion, search and insertion performances will not become worse in the *MVR**-tree.

3.4 Construction of The MVR*-tree

The *MVR**-tree is an improved *MVR*-tree with an array of active object identifier (*AOI*). *AOI* indicates which objects in a leaf are active at the version. This allows the multiple versions of R-trees in an *MVR*-tree can share a leaf. Therefore, in the *MVR**-tree, a leaf is copied only when the leaf is full and an object is inserted in the leaf. The algorithms of the *MVR**-tree can be constructed by modifying the algorithm of the *MVR*-tree such that it makes *AOI* effective. The insertion algorithm of the *MVR**-tree is presented as follows.

(a) Insert an object

InsertInMVR*(V_n, O_x)
V_n: R-tree version
O_x : Object
/* insert object O_x into version V_n */
S1: Let the root of V_n be R.
S2: Tracing from R to lower nodes, find *AOI* E pointing to leaf L in which O_x should be inserted.
S3: Copy the nodes on the path from R to E that are not in V_n into V_n.
S4: If E is not in Vn, make a copy of AOI E and set the pointer of the copied AOI to L. Set E to the copied AOI.
S5: If L is not full, insert O_x into L and set 1 to the corresponding element in E. Return.
S6: Otherwise, make copy L1 of L and copy E1of E. Remove objects in L1 that are not active at this version.
S7: If L1 is full, create new leaf L2 and copy E1 of E, then move (C x **M'**) objects in L1 to L2. Insert O_x by the R-tree's algorithm. Adjust AOIs pointing to L1 and split leaf.

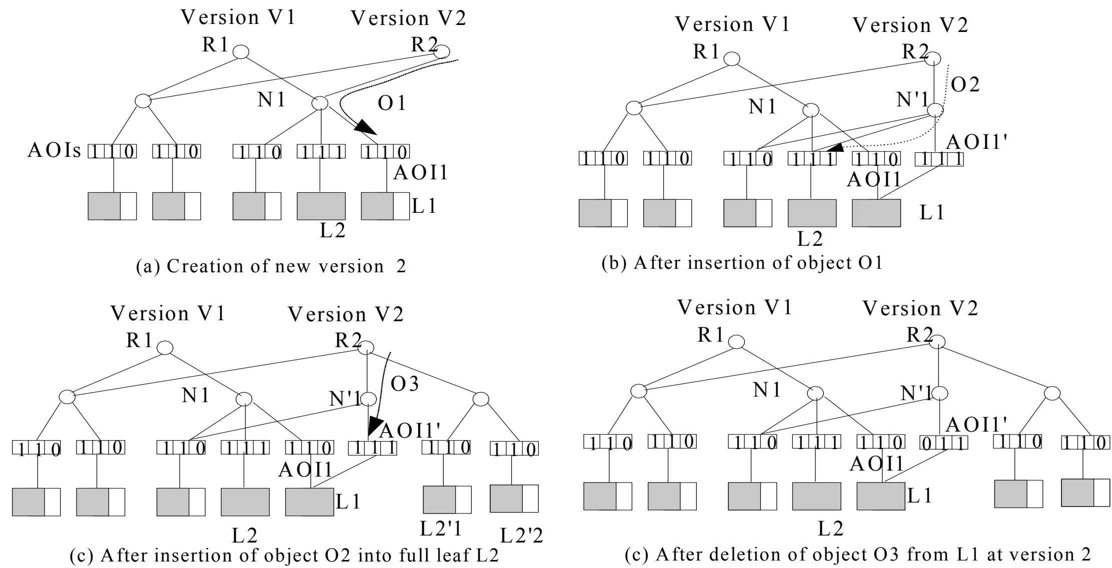


Fig.5 The MVR*-tree managing versions

(b) Delete an object
In the deletion procedure, an object is not removed from a leaf immediately, but a corresponding element of AIO is set to be 0. An element with 0 means the corresponding object in a leaf is not active in a version.

DeleteFromMVR*(V_n, O_x)
Vn: R-tree version
O_x : Object
/* delete object O_x from V_n. */
S1: Let the root of V_n be R.
S2: Tracing from R, find AOI E pointing to leaf L in which O_x exists.
S3: If L is in V_n, remove O_x from L and set value 0 to the corresponding element of E If L satisfies the underflow condition, apply the same algorithm of the R-tree. Return.
S4: If E is not in V_n, copy the nodes and AOI on the path from R to E that are not in Vn into Vn. E<-copied AOI.
S5: If L is in V_n, remove O_x from L.
S6: Set 0 to the corresponding element of E.
S7: If L' satisfies the underflow condition, remove L' from Vn and adjust the tree. Then re-insert the objects in L' .These procedures are the same as those of the R-tree.

Fig.5 shows insertion and deletion processes of an *MVR**-tree. At the first insertion of object **O1** into version **V2** in Fig.5 (a), since leaf **L** is not full, object **O1** is inserted in **L1** through *AOI 1*. Bit 1 in *AOI* indicates that the object at the position in the corresponding leaf is active at the version. *AOI 1* belongs to version **V1**. Therefore, a copy of *AOI 1*, *AOI 1'*, pointing to **L1** is created and **O1** is inserted in **L1**, as shown Fig.5 (b). *AOI 1* in **V1** and *AOI 1'* in **V2** indicate which objects are active in each version of trees. That is, the third object in L1 is not active in version **V1**, but active in version **V2**. When object **O2** is inserted full leaf **L2** in Fig.5 (b), a copy of **L2**, **L2'1**, and a new leaf, **L2'2**, are created. Two objects in **L2'1** are moved to **L2'2**, and **O2** is then inserted. Leaves **L2'1** and **L2'2** are added to node **N1'** as children. However, a node cannot have more than 3 nodes. As a result, new node is created to manage **L2'1** and **L2'2**, and the new node is added to root **R2** as a child. The *MVR**-tree in Fig.5 (c) is obtained. When object **O3** that is stored as the first object in leaf **L1** is deleted at version **V2**, **O3** is not removed from **L** but the corresponding element (the first element) of *AOI 1'* pointing to **L1** in version **V2** is set to be 0, as shown in Fig.5 (d).

4. EXPERIMENTAL RESULTS

To evaluate the performances and characteristics of the *MVR*-tree, the *MVR**-tree, and the simple R-tree method, a series of simulation tests is carried out. In the simple R-tree method, R-trees managing the versions are created to manage the multi-version drawings. An R-tree corresponding to a new version is created by copying an old version R-tree and modifying the contents. The roots of these R-trees are managed in the database. Hereinafter we represent this method the R-tree.

(1) Conditions of Experiments
The *MVR* and *MVR** trees are expansions of the R-tree. The R-tree with the quadratic split algorithm is used as a basic structure in both cases of the *MVR* and *MVR** trees. The number of maximum child nodes the R-tree is equal to 3, and the number of objects in a leaf is 20. The original version drawing, v0.0, with 100,000 objects is prepared. The entire data space is a 10,000x10,000 square area. Objects' positions are generated randomly from a uniform distribution. Object are rectangles with the sides varying from 5 to 10 in length.

Versions, v1.0, v2.0,…, and v6.0, are then created from version v0.0. Vi+1.0 is created from vi.0. 10,000 objects are added in each new version in any case. The following two cases are tested.

Case 1: 10,000 objects are inserted into a 100x100 square area at the center of the entire data space.

Case 2: 10,000 objects are inserted into the entire area randomly.

(2) Experimental Results

Table 1 shows the construction time of the R-tree, *MVR*-tree and *MVR**-tree with 7 versions in Case 1. The R-tree method requires more construction time than others. It takes almost the same time to construct the *MVR* and *MVR** trees, because the R-tree methods requires much more nodes and leaves than the *MVR* and *MVR** trees. As for Case 2 data sets, the construction time for the R-tree is also longer than the *MVR* and *MVR** trees.

The numbers of nodes and leaves in the R, *MVR*, and *MVR** trees in Case 1 experiment are shown in Fig.6 (a). The number of total nodes in the R-tree method is proportional to the number of versions. An R-tree with only one version contains approximately 5.5K nodes and the total nodes of the R-trees with 7 versions are approximately 48K. On the other hand, the number of total nodes in the *MVR* and *MVR** trees are approximately 13.5K. The numbers of total leaves in the R, *MVR*, and *MVR** trees are 62.6K, 17.9K, and 14.1K, respectively. Namely, the required storages for the *MVR*, and *MVR** tree are reduced to 29% and 23% of the R-tree,, when the objects are inserted in a small area.

The results of Case 2 experiment are shown in Fig.6 (b). Regarding the R-tree, the numbers of total nodes and leaves are almost equal to those of Case 1. However, the *MVR*-tree contains more nodes and leaves compared to Case 1. Especially, the number of leaves in the *MVR*-tree is 80% of the R-tree in Case 1, because objects are inserted into almost leaves. In the *MVR**-tree, the number of leaves is reduced to 1/3 of the *MVR*-tree. These results reveal that the *MVR**-tree requires much less storage than the *MVR*-tree when the modification is applied in a wide area. Even if the modification is local, the number of leaves nodes in the MVR*-tree is less than that of the MVR-tree. The search performances of these methods in Case 2 are almost the same as shown in Table 2, because the structural properties of these methods are identical. Although additional structure called AOI was introduced in the MVR*-tree, the search performances do not deteriorate.

Table 1 Construction results of R, MVR, and MVR*trees in Case 1

	Time(sec.)	Nodes	Leaves
R-tree	73.2	46,899	62,608
MVR-tree	32.8	13,551	17,877
MVR*-tree	33.7	13,551	14,129

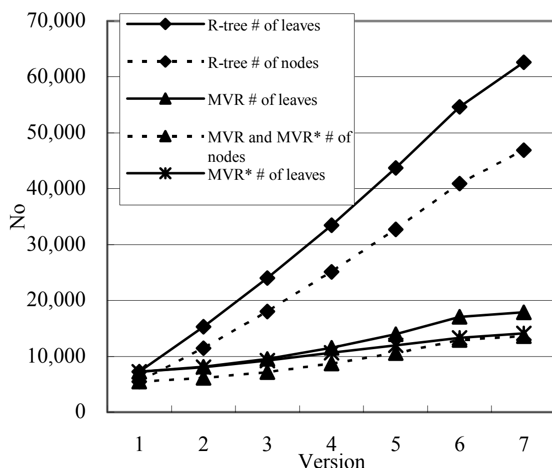
Table 2 Performances of the nearest neighbor search in Case 2

	Time(msec.)	Visited nodes
R-tree	0.078	18.72
MVR-tree	0.078	18.72
MVR*-tree	0.079	18.72

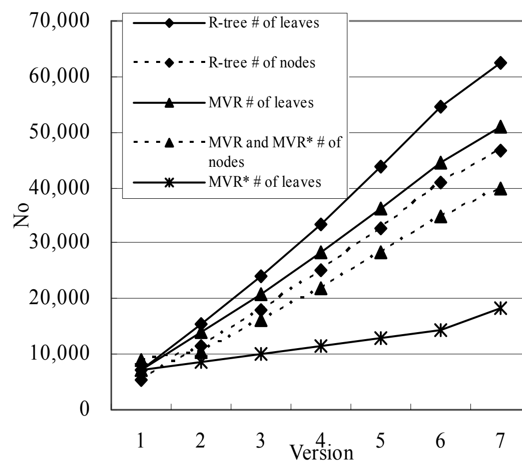
These experiments reveal that the *MVR**-tree is much efficient than the simple R-tree or *MVR*-tree in the amount of storage, because the R-trees in MVR*-tree share the nodes and leaves effectively by introducing the AOI structure.

5. CONCLUSION

We proposed new data structures, the *MVR*-tree and *MVR**-tree, for handling sets of spatial objects such as multiple versions of an engineering drawing, diagram, and so on. The experimental results reveal the better performances of the *MVR* and *MVR** trees than the simple R-tree method when a modification is applied to a small portion of the data space. The number of nodes and leaves in the *MVR*-tree is 29% of the R-tree. When a modification affects a wide portion of the data space, however,



(a) Case 1: Modifications are applied to a small area.



(b) Case 2: Modifications are applied a wide area.

Fig.5 The numbers of nodes and leaves of the R, MVR and MVR* tree for 6 versions

the number of nodes and leaves in the MVR-tree decreases only 17% compared to the R-tree. To improve this, the *MVR**-tree was developed. In the *MVR**-tree, the number of nodes and leaves is 25% of the R-tree, in the case of the modification is local and is 53% in the case of the modification is applied wide area.

Since the engineering database is required to manage a large number of multi-version drawings efficiently, we plan to apply the *MVR**-tree method as an index structure for engineering databases.

References

[1]R. H. Katz, “Toward a Unified Framework for Version Modeling in Engineering Databases,” **ACM Computing Surveys**, Vol. 22, No. 4, pp.375-408(1990).

[2] V. J. Tsotras and B. Gopinath, “Effient Algoruthms for Managing The History of Evolving Databases,” **Proc. Int. Conf. Database Theory**, pp.141-174(1990).

[3] S. Lanka and E. Mays, “Fully persistent B+-trees,” **Proc. ACM SIGMOD Conf**., pp.426-435(1991).

[4] D. Lomet and B. Salzberg, “Access Methods for Mutiversion Data,” **Proc. ACM SIGMOD conf**., pp.315-324(1989).

[5] P. J. Varman and R. M. Verma, “An Efficient Multiversion Access Structure,” **IEEE trans. KDE**, Vol. 9, No. 3, pp.391-409(1997).

[6] A. Guttman, “R-trees: A Dynamic Index Structure for Spatial Searching,” **Proc. ACM SIGMOD**, pp.47-57(1984).

[7] Y. Nakamura, et al., "A Balanced Hierarchical Data Structure for Multidimensional Data with Efficient Dynamic Characteristics,” **IEEE trans. KDE**, Vol.5, No.4, pp.682-694(1993).

[8] V. J. Tsotras and B. Gopinath: “Efficient Algorithm,s for Manageing the Histrory of Evoling Databases,” Proc. Int. Conf. Database Theory, pp. 141-174(1990).

[9] G. Kollis, D. Gunopulos and V. J. Tsotras: “On Indexing Mobile Objects, Proc. PODS, pp.261-272(1999).

[10] Y. Nakamura, H. Dekihara, and R. Furukawa: “Spatio-temporal Data Management for Moving Objects Using the PMD-tree,” Lecture Notes in Computer Science 1552, Advances in Database Technology(Y. Kanbayashi, D. L. Lee et. al (Eds)), pp.496-507 (1999).