M-tree (Version 0.911) User's Guide

Marco Patella
DEIS - CSITE-CNR
University of Bologna, Italy
mpatella@deis.unibo.it

December 19, 2000

Abstract

M-tree [CPZ97] is a dynamic access structure suitable to index generic "metric spaces", where the function used to compare objects is a metric, satisfying the positivity, symmetry, and triangle inequality postulates. Our M-tree implementation is based on the Generalized Search Tree (GiST) C++ package [HNP95], a structure extendible both in the data types and in the queries it can support. This document serves as documentation for the user who wants to understand the features provided by this software, and as reference for the user willing to tailor the M-tree for his/her application.

Contents

1 Introduction

The M-tree is an index structure able to organize and search large data sets drawn from a generic "metric space" $\mathcal{M} = (\mathcal{D}, d)$, where $\mathcal{D}$ is a domain of feature values, and $d : \mathcal{D}^2 \to \Re_0^+$ is a metric function satisfying the following axioms, for each $v_x, v_y, v_z \in \mathcal{D}$:

$$
\begin{aligned}
d(v_x, v_x) &= 0 && \text{(non-negativity)}\\
d(v_x, v_y) &> 0 \qquad (v_x \neq v_y) && \text{(positivity)}\\
d(v_x, v_y) &= d(v_y, v_x) && \text{(symmetry)}\\
d(v_x, v_z) + d(v_z, v_y) &\geq d(v_x, v_y) && \text{(triangle inequality)}
\end{aligned}
$$

The overall structure of the M-tree is introduced in [CPZ97], its extension to deal with *complex queries* can be found in [CPZ98], and a bulk-loading algorithm is presented in [CP98]. Finally, a more comprehensive description of the structure, together with a discussion about the underlying principles and about possible extensions, can be found in [Pat99].

Our implementation is based on the GiST C++ package [HNP95], and the version numbering follows that of GiST libraries (this is why numbering of releases started from 0.9). In order to successfully build the M-tree project, the user needs to obtain the GiST library having a version number (most) equal to that of the M-tree version.[1]

In the next Section, we illustrate the basic features provided by the M-tree software library. In Section 3, we present a brief description of all the classes included in the library, showing the user how to tailor the index structure to his/her needs and how to tune index performance. Finally, in Section 4, we describe the contents of each file included in the library.

2 Features

In order to define the data that will be indexed by the M-tree the user has only to re-define the `Object` class (see Section 3.2). An additional degree of freedom provided by the library is the chance that the user is given to specify the size of each node stored on disk. This, of course, affects the nodes' fan-out and, therefore, the index performance.

Creating a new M-tree is as easy as saying "new"! In fact, in order to create a new index structure, it is sufficient to instantiate the `MT` class, e.g.:

```
MT *tree=new MT;
```

Then, in order to provide persistence to the index, the user has to "link" the `MT` object to a file on disk, with the following instructions:

```
tree->Create(filename);
```

[1] In this case, GiST library version 0.9, also available at URL: `http://www-db.deis.unibo.it/~patella/MMindex.html`.

if the index has to be created from scratch, or:

```
tree->Open(filename);
```

if the index is already stored in the file named `filename`. Objects can be inserted in the tree one by one or using a bulk-loading technique (see Section 3.1). The M-tree can be searched using nearest neighbors and range queries, even in a "complex" environment [CPZ98] where query predicates are expressed as conjunctions, disjunctions, and negations of simple predicates (see Section 3.9). Moreover, a sorted access to the tree is also provided, where indexed objects are returned one by one sorted by increasing distance to the query predicate.

The library includes three integer values, `compdists`, `IOread`, and `IOwrite`, to keep track of CPU and I/O costs. More precisely, every time a distance is computed (i.e. the method `Object::distance()` is called), the value of `compdists` is increased by 1; every time a node is read from disk (through the method `MTfile::Read()`), the value of `IOread` is increased by 1; every time a node is written on disk (through the method `MTfile::Write()`), the value of `IOwrite` is increased by 1.

3 M-tree Classes Reference

This Section describes all the classes included in the M-tree library. For each class, the following items are listed:

- The name of the super-class (if applicable)
- The name of the sub-classes (if applicable)
- The name and a brief description of attributes added to those of the super-class
- The name and the signature of methods the user has to redefine for the class
- The name and the signature of public methods provided by the class
- The name and the signature of private methods used by the class

3.1 `MT` Class

Class name: `MT`

Super-class: `GiST`

Sub-classes:

Attributes:

Public methods:

- `GiSTlist<MTentry *> RangeSearch(const MTquery& query) const`: range search algorithm
- `MTentry **TopSearch(const TopQuery& query) const`: top (nearest neighbors) search algorithm
- `void BulkLoad(MTentry **data, int n, double padFactor, char *name)`: bulk-loading algorithm
- `int MaxLevel() const`: returns the tree height
- `MTnode *ParentNode(MTnode *node) const`: returns the parent node of `node`
- `BOOL CheckNode(GiSTpath path, MTentry *e) const`: debug function, checks for node consistency

Other methods:

- `GiSTlist<char *> *SplitTree(int *ncreated, int level, GiSTlist<MTentry *> *children, char *name)`: splits a tree into a list of `MT`s, used during bulk-load
- `void Append(MTnode *to, MTnode *from)`: appends the sub-tree rooted at `from` to node `to`, used during bulk-load
- `void AdjKeys(GiSTnode *node)`: redefined method from class `GiST` to include computation of distances from the parent object

This class implements the M-tree index structure; it is a sub-class of the basic `GiST` class. The class definition can be found in the file `MT.h`, while its implementation is included in the files `MT.cpp` and `Bulkload.cpp`, the latter containing the code for the bulk-loading algorithm [CP98]. Objects of the `MT` class can be created as usual, and linked to a file stored on disk by using the `Open` method of the `GiST` class, providing the file name. As an example:

```
MT *tree=new MT;
tree->Open("index.M3");
...  // use of the tree
delete tree;
```

Insertion of `Object`s in the tree can be performed using two different modalities:

- by inserting objects one-by-one, using the `Insert` method, or
- by using the `BulkLoad` method.

When inserting objects one-by-one, the user has to specify, for each insertion, an `MTentry` representing the `Object`, given as `MTentry(MTkey(obj, 0, 0))`, and an integer number representing the identifier of the object, which should be unique in the data-set. A simple insertion could be performed by way of the following instruction:

```
tree->Insert(MTentry(MTkey(obj,0,0)),17);
```

This inserts the `Object obj` in the tree with identifier equal to 17.

In order to specify the split policy [CPZ97], the following variables, defined in `MTnode.cpp`, should be set:

`double MIN_UTIL` Minimum node utilization. It is used to guarantee a minimum fill factor for tree nodes during the split. It can assume values in the range $[0 \div 0.5)$.

`pp_function PROMOTE_PART_FUNCTION` Promotion method. Specifies the algorithm used to promote objects in the parent role. Assuming the set of following values:

- `RANDOM` random promotion;
- `CONFIRMED` confirmed promotion; variable `PROMOTE_VOTE_FUNCTION` is then used to choose between confirmed policies;
- `MAX_UB_DIST` maximum upper bound on distances policy (i.e. the two objects having the maximum distance from the parent object are chosen);
- `MIN_RAD` minimum maximum radius policy;
- `MIN_OVERLAPS` minimum overlap policy;
- `SAMPLING` sampling promotion; variable `NUM_CANDIDATES` specifies the number of samples.

`pv_function PROMOTE_VOTE_FUNCTION` Confirmed promotion method. Specifies the algorithm used to promote one object as one of the two parents, the other being the parent object of the split node. It is used only if `PROMOTE_PART_FUNCTION=CONFIRMED`. Assuming the set of following values:

- `RANDOMV` random confirmed promotion;
- `SAMPLINGV` sampling confirmed promotion; variable `NUM_CANDIDATES` specifies the number of samples;
- `MAX_LB_DIST` maximum lower bound on distances promotion (i.e. the object farthest from the parent object is chosen);
- `mM_RAD` minimum radius confirmed policy, variable `RADIUS_FUNCTION` is then used to choose between available policies.

`r_function RADIUS_FUNCTION` Minimum radius method. Assuming the set of following values:

- `LB` minimum maximum lower bound on radius policy;
- `AVG` minimum maximum average bound on radius policy;
- `UB` minimum maximum upper bound on radius policy.

`pp_function SECONDARY_PART_FUNCTION` Root promotion method. It is only used when splitting the root node and can assume the same values of the `PROMOTE_PART_FUNCTION` variable. However, since the root node does not have a parent object, this cannot be a confirmed policy.

`int NUM_CANDIDATES` Number of candidate objects for sampling methods.

`s_function SPLIT_FUNCTION` Split function. This specifies the way objects in the overfull node are to be divided between the two new nodes. Assuming the set of following values:

- `G_HYPERPL` the generalized hyperplane partition strategy;
- `BAL_G_HYPERPL` the balanced hyperplane partition strategy;
- `BALANCED` the balanced strategy.

Of course, both the `BAL_G_HYPERPL` and the `BALANCED` methods obtain perfectly balanced nodes, whereas with the `G_HYPERPL` strategy the user can tune the node utilization by modifying the value of the `MIN_UTIL` variable. We suggest to use the `G_HYPERPL` strategy, since a balanced split tends to produce high covering radii, and this negatively affects index performance during the search. Having unbalanced nodes ”only” increases the size of the tree (i.e. the number of nodes), but most of the nodes will be pruned when searching the M-tree.

Split policies not detailed in [CPZ97] can be found in [Pat99].

The bulk-loading algorithm [CP98] is implemented by the `BulkLoad` method. Arguments of the `BulkLoad` algorithm are:

`MTentry **data` This is the array containing all the entries to be inserted in the tree. Each entry is given an identifier equal to the position of the entry in the array.

`int n` The number of entries in the `data` array.

`double padFactor` This factor, assuming values in the range $[0 \div 1]$, specifies the maximum node utilization of the tree (can be lower than 1 if the user wants to insert other objects in the tree).

`char *name` The name root for the files storing the temporary indices on disk. They will be deleted upon completion of the method.

The minimum node utilization can be specified by the user through the MIN_UTIL variable.

The RangeSearch method implements the range search algorithm: when called for the tree with an arbitrarily complex MTquery (see Section 3.9), it recursively descends the tree, returning a list containing all entries that are consistent with (i.e. satisfy) the query predicate. Returned entries contain the distance between the entry object and the query predicate as the maximum radius of the entry (accessible through the maxradius() method). The typical way to resolve a range query is the following:

```
MT *tree=new MT;
tree->Open(...); // tree construction
...
MTquery query(...); // query construction
GiSTlist<MTentry *> list=tree->RangeSearch(query);
while(!list.IsEmpty()) {
    MTentry *e=list.RemoveFront();
    Object o=e->object(); // result object
    double dist=e->maxradius(); // distance from query predicate
    ...  // result evaluation
    delete e;
}
```

The top search algorithm is implemented in the TopSearch method. Given a TopQuery (see Section 3.9), the algorithm returns an array containing the k entries nearest to the query predicate, sorted by increasing distance. The distance between each entry and the query predicate is returned as the maximum radius of each MTentry (see Section 3.5). The classical way to resolve a top query is the following:

```
MT *tree=new MT;
tree->Open(...); // tree construction
...
TopQuery query(...); // query construction
MTentry **list=tree->TopSearch(query);
for(int i=0; i<query.k; i++) {
    MTentry *e=list[i];
    Object o=e->object(); // result object
    double dist=e->maxradius(); // distance from query predicate
    ...  // result evaluation
    delete e;
}   delete []list;
```

3.2 Object Class

Class name: Object

Super-class: `GiSTobject`

Sub-classes:

Attributes: User-defined

User-defined methods:

- `Object()`: default constructor
- `Object(const Object& obj)`: copy constructor
- `Object(char *key)`: member constructor
- `~Object()`: destructor
- `Object& operator=(const Object& obj)`: assignment operator
- `int operator==(const Object& obj) const`: equality operator
- `double distance(const Object& other) const`: distance method
- `void Compress(char *key)`: compress method
- `int CompressedLength() const`: size of compressed object

Public methods:

- `int operator!=(const Object& obj) const`: inequality operator

Other methods:

Instances of this class are the objects indexed by the M-tree. This is the only class in the whole project that needs a redefinition according to the application defined by the user. The declaration of the class is in the file `MTobject.h`, whereas class implementation can be found in the file `MTobject.cpp`.

When the user redefines the `Object` class, he/she has to provide all the information necessary to compute the distance between two `Objects`. Attributes of the `Object` class, therefore, are represented only by such information (e.g. in the vector space example, the only attribute of the `Object` class is the array of the coordinates). Besides, the following methods should be redefined, in order to obtain the desired behavior for the indexed objects:

`Object()` This is the default constructor. It should assign default values to all object attributes.

`Object(const Object& obj)` This is the copy constructor. It should provide a suitable way to copy the values of all attributes from `Object obj` to `Object *this`.

`Object(char *key)` This is a member constructor needed to "uncompress" an `Object` from the node representation stored on disk. This method is called in the `Decompress` method of the `MTentry` class (see Section 3.5). Basically, this should perform the inverse operation with respect to the `Compress` method.

~`Object()` This is the class destructor. It should de-allocate the memory possibly allocated during `Object` construction.

`Object& operator=(const Object& obj)` This is the assignment operator. It should basically mimic the code of the destructor and of the copy constructor methods.

`int operator==(const Object& obj) const` This is the equality operator. It should return a non-zero value iff the two `Object`s can be assumed as equal (or equivalent). An alternative way to define such method is to return a non-zero value iff the `distance` (see below) between the two `Object`s is equal to 0.

`double distance(const Object& other) const` This is the distance method. It is, of course, the basic method for the M-tree index structure.

`void Compress(char *key)` This is the compress method, which should copy all the attributes of the `Object` into the `key` buffer. It is used in the `Compress` method of the `MTentry` class (see Section 3.5).

`int CompressedLength() const` This is the size of the compressed `Object`, i.e. the sum of the sizes (in bytes) of each attribute as saved in the node representation stored on disk (see method `Compress`).

Other functions that need to be redefined are the following:

`double maxDist()` This function defines the maximum possible value for the distance between two `Object`s.

`int sizeofObject()` This function returns the size of each compressed `Object`, in case `Object`s have all the same size, or 0 if `Object`s have different size, so that execution of the `CompressedLength()` method is needed to retrieve it.

`Object *Read()` This function returns an `Object` as read from standard input.

`Object *Read(FILE *fp)` This function returns an `Object` as read from file `fp`.

3.3 `MTkey` Class

Class name: `MTkey`

Super-class: `GiSTobject`

Sub-classes:

Attributes:

- `Object obj`: private, indexed object

- `double rmin`: private, minimum radius (only used for compatibility with complex applications, e.g. see [CA97])
- `double rmax`: private, maximum radius (i.e. the entry radius)
- `double distance`: public, distance between the `Object` and the parent `Object`
- `BOOL splitted`: public, indicates that this child node has been split
- `BOOL recomp`: public, indicates that this child node is in an inconsistent state (only used during node splitting)

Public methods:

Other methods:

- `MTkey()`: default constructor
- `MTkey(const MTkey& k)`: copy constructor
- `GiSTobject *Copy() const`: copy operator, returns a new `MTkey` as a copy of the current one
- ∼`MTkey()`: destructor
- `MTkey& operator=(const MTkey& k)`: copy operator
- `int operator==(const MTkey& k) const`: equality operator
- `int overlap (const MTkey& k) const`: overlap operator, indicates if two regions of the metric space overlap
- `int contained(const MTkey& k) const`: containment operator, indicates if the current region is contained in the given one
- `int contains(const MTkey& k) const`: containment operator, indicates if the given region is contained in the current one
- `MTkey *expand(const MTkey& k)`: expansion operator, expands the current region in order to accommodate the given region

This is a service class which should never be istantiated nor referred by the user. It contains the representation of the metric space region associated to each entry of the M-tree. Since the M-tree uses balls to organize the space, each region is defined by an `Object` (the center of the region) and a radius. Definition and implementation of this class are included in the file `MTentry.h`.

3.4 `MTpenalty` Class

Class name: `MTpenalty`

Super-class: `GiSTpenalty`

Sub-classes:

Attributes:

- `double distance`: distance to the parent `Object`

Public methods:

Other methods:

- `MTpenalty()`: default constructor
- `MTpenalty(const double v, const double d)`: member constructor

This is a service class which should never be istantiated nor referred by the user. It is a redefinition of the basic Penalty class of GiST and it is used when inserting objects in the M-tree. Its definition and implementation are included in the file `MTentry.h`.

3.5 `MTentry` Class

Class name: `MTentry`

Super-class: `GiSTentry`

Sub-classes:

Attributes:

Public methods:

- `MTentry()`: default constructor
- `MTentry(const MTentry& e)`: copy constructor
- `GiSTobject *Copy() const`: copy operator, returns a new `MTentry` as a copy of the current one

Other methods:

- `MTentry(const MTkey& k, const GiSTpage p)`: member constructor
- `void InitKey()`: re-initialization of the `MTkey`
- `int IsEqual(const GiSTobject& obj) const`: equality operator
- `GiSTpenalty *Penalty(const GiSTentry& newEntry) const`: penalty computation
- `GiSTpenalty *Penalty(const GiSTentry& newEntry, MTpenalty *minPenalty) const`: penalty computation, redefinition to optimize distance computations

- `int CompressedLength() const`: size of the compressed `MTentry`
- `GiSTcompressedEntry Compress() const`: compression method, used when storing a node on disk
- `void Decompress(const GiSTcompressedEntry entry)`: decompression method, used when retrieving a node from disk
- `int Compare(const GiSTentry& entry) const`: comparison method, used when ordering the entries in a node

Instances of the `MTentry` class are the entries contained in each node and represent nodes at the next lower level for internal nodes, and objects for leaf nodes. The definition of the class is in the file `MTentry.h`, whereas its implementation can be found in file `MTentry.cpp`.

3.6 `MTfile` Class

Class name: `MTfile`

Super-class: `GiSTfile`

Sub-classes:

Attributes:

User-defined methods:

- `int PageSize() const`: size of disk pages in bytes

Public methods:

Other methods:

- `void Create(const char *filename)`: creates a new index file on disk
- `void Open(const char *filename)`: opens an existing index file
- `void Close()`: closes an open index file
- `void Read(GiSTpage page, char *buf)`: reads a node from an open index file
- `void Write(GiSTpage page, const char *buf)`: writes a node to an open index file
- `GiSTpage Allocate()`: allocates a new page on an open index file
- `void Deallocate(GiSTpage)`: deletes an existing page from an open index file

This is a redefinition of the basic `GiSTfile` class in order to do some book-keeping, like counting disk access requests, etc. File `MTfile.h` contains the definition of the class, whereas the class implementation is in file `MTfile.cpp`. The user can modify the size in bytes of each disk page (through the `PageSize()` method) in order to tailor nodes fanout for his/her application.

3.7 MTnode Class

Class name: `MTnode`

Super-class: `GiSTnode`

Sub-classes:

Attributes:

- `Object *obj`: parent object

Public methods:

- `GiSTlist<MTentry *> RangeSearch(MTquery& query) const`: range search algorithm

Other methods:

- `MTnode()`: default constructor
- `MTnode(const MTnode& node)`: copy constructor
- `GiSTobject *Copy() const`: copy operator
- `GiSTobject *NCopy()`: alternate copy operator, used during node split
- `GiSTpage SearchMinPenalty(const GiSTentry& entry) const`: redefinition of the basic `GiSTnode` method to optimize distance computations
- `void InsertBefore(const GiSTentry& entry, int index)`: redefinition of the basic `GiSTnode` method to insert entries in order of increasing distance with respect to the parent `Object`
- `GiSTnode *PickSplit()`: redefinition of the basic `GiSTnode` method to implement the node split logic
- `GiSTentry *Union() const`: redefinition of the basic `GiSTnode` method to compute the region associated with each node
- `MTnode *PromotePart()`: parent `Object` promotion
- `MTnode *PromoteVote()`: parent `Object` confirmed promotion
- `int *PickCandidates() const`: picks a sample of `Objects` as candidates for promotion

- `void Split(MTnode *node, int *leftvec, int *rightvec, int *leftdeletes, int *rightdeletes)`: performs the split
- `int IsUnderFull(const GiSTstore& store) const`: checks node underfilling

The `MTnode` class implements the logic for inserting an entry in a node and for node splitting. The definition of the `MTnode` class is included in the file `MTnode.h`, while the implementation can be found in the file `MTnode.cpp`.

3.8 `MTpred` Class

Class name: `MTpred`

Super-class: `GiSTobject`

Sub-classes: `Pred, AndPred, OrPred, NotPred`

Attributes:

Public methods:

- `double distance(const Object& obj) const`: pure virtual, distance method

Other methods:

- ~`MTpred()`: destructor

This is the base class for predicates supported by M-tree. It is a virtual class and complex predicates [CPZ98] for the fuzzy language can be defined by way of the sub-classes `Pred`, `AndPred`, `OrPred`, and `NotPred` (see below). If the user wishes to define other complex predicates, such as those created by using the "weighted sum" language [CPZ98], he/she has to derive sub-classes from the `MTpred` class, and to compose predicates using the `Pred` class. The definition and the implementation of the class, together with those of the sub-classes, are included in the file `MTpred.h`.

3.8.1 `Pred` Class

Class name: `Pred`

Super-class: `MTpred`

Sub-classes:

Attributes:

- `Object object`: private, object of the simple predicate

Public methods:

- `Pred(const Object& obj)`: member constructor
- `Pred(const Pred& p)`: copy constructor
- `GiSTobject *Copy() const`: copy operator
- `const Object& obj()`: access to private member `object`

Other methods:

Objects of this class are simple predicates, i.e. leaves of the predicate tree. Each `Pred` contains the reference to a simple `Object`, and the distance between the predicate and an object is simply computed as the distance between the two `Objects`.

3.8.2 `AndPred` Class

Class name: `AndPred`

Super-class: `MTpred`

Sub-classes:

Attributes:

- `MTpred *pred1, *pred2`: private, sub-predicates

Public methods:

- `AndPred(const MTpred *p1, const MTpred *p2)`: member constructor
- `GiSTobject *Copy() const`: copy operator
- `MTpred *Pred1()`: access to private member `pred1`
- `MTpred *Pred2()`: access to private member `pred2`

Other methods:

- `~AndPred()`: destructor

This class defines the conjunction between two sub-predicates. The semantics of the conjunction is given through the `language query_language` variable, assuming one of the following values:

- `FUZZY_STANDARD` fuzzy standard language
- `FUZZY_ALGEBRAIC` fuzzy algebraic language

and the two functions:

- `double Dist2Sim(double dist)` transforms distance values into similarity scores (correspondence function h)

- `double Sim2Dist(double sim)` transforms similarity scores into distance values (inverse of the correspondence function h, h^{-1})

Implementation of the `Dist2Sim` and `Sim2Dist` functions is given in the `MTpred.cpp` file.

3.8.3 `OrPred` Class

Class name: `OrPred`

Super-class: `MTpred`

Sub-classes:

Attributes:

- `MTpred *pred1, *pred2`: private, sub-predicates

Public methods:

- `OrPred(const MTpred *p1, const MTpred *p2)`: member constructor
- `GiSTobject *Copy() const`: copy operator
- `MTpred *Pred1()`: access to private member `pred1`
- `MTpred *Pred2()`: access to private member `pred2`

Other methods:

- `~OrPred()`: destructor

This class defines the disjunction between two sub-predicates. The semantics of the disjunction is given in the same way of the conjunction.

3.8.4 `NotPred` Class

Class name: `NotPred`

Super-class: `MTpred`

Sub-classes:

Attributes:

- `MTpred *pred`: private, sub-predicate

Public methods:

- `NotPred(const MTpred p)`: member constructor
- `GiSTobject *Copy() const`: copy operator
- `MTpred *Pred()`: access to private member `pred`

Other methods:

- ~`NotPred()`: destructor

This class defines the negation of a predicate. The distance between an `Object o` and a negated predicate `p` is given as:

```
Dist2Sim(1-Sim2Dist(p->Pred()->distance(o)))
```

3.9 `MTquery` Class

Class name: `MTquery`

Super-class: `GiSTobject`

Sub-classes: `SimpleQuery`

Attributes:

- `double grade`: private, similarity score computed for the currently examined `MTentry` with respect to the query predicate
- `BOOL isOpen`: private, indicates if the query is open

Public methods:

- `MTquery()`: default constructor
- `MTquery(const double g, const BOOL o=FALSE)`: member constructor
- `int Consistent(const GiSTentry& entry)`: pure virtual, checks for consistency
- `int NonConsistent(const GiSTentry& entry)`: pure virtual, checks for non-consistency (useful for negated queries)
- `double Grade() const`: access to private member `grade`
- `double SetGrade(double p_grade)`: access to private member `grade`

Other methods:

- ~`MTquery()`: destructor

This is the base class for range queries supported by M-tree. The definition and the implementation of the class are included in the file `MTpred.h`. The `MTpred.h` file also contains the definition of classes `AndQuery`, `OrQuery`, and `NotQuery`. However, such classes are included only for future expansions and should not be instantiated.

3.9.1 `SimpleQuery` Class

Class name: `SimpleQuery`

Super-class: `MTquery`

Sub-classes:

Attributes:

- `MTpred *pred`: private, query predicate
- `double radius`: private, query radius

Public methods:

- `SimpleQuery(const MTpred *p, const double r, const BOOL o=FALSE)`: member constructor
- `SimpleQuery(const SimpleQuery& q)`: copy constructor
- `GiSTobject *Copy() const`: copy operator
- `double Radius() const`: access to private member `radius`
- `void SetRadius(double p_radius)`: access to private member `radius`

Other methods:

- `~SimpleQuery()`: destructor

This class implements the simple range query. Its definition is included in the file `MTpred.h`, while the implementation is in file `MTpred.cpp`. It is used in the `RangeSearch` method of the class `MT` (see Section 3.1). If the query is open (`o=TRUE` when constructing the `SimpleQuery` object), only objects whose distance to the query predicate is lower than the provided radius are returned as the result; otherwise (`o=FALSE`), also objects whose distance to the query predicate is equal to the provided radius are returned.

3.9.2 `TopQuery` Class

Class name: `TopQuery`

Super-class: `GiSTobject`

Sub-classes:

Attributes:

- MTpred *pred: private, query predicate
- int k: public, number of neighbors

Public methods:

- TopQuery(const MTpred *p, const int n): member constructor
- TopQuery(const TopQuery& q): copy constructor
- GiSTobject *Copy() const: copy operator
- const MTpred *Pred() const: access to private member pred

Other methods:

- ~TopQuery(): destructor

This class implements the simple top query. Its definition and implementation are included in the file MTpred.h. It is used in the TopSearch method of the class MT (see Section 3.1).

3.10 MTcursor Class

Class name: MTcursor

Super-class:

Sub-classes:

Attributes:

- const MT& tree: the tree the cursor acts upon
- list<MTentry *> results: the queue of partial results
- list<dst *> queue: the queue of active nodes
- MTpred *query: the query predicate

Public methods:

- MTcursor(const MT& tree, const MTpred& query): member constructor
- MTentry *Next(): fetches next result from the queue
- BOOL IsReady() const: returns TRUE iff it is not necessary to fetch nodes to return the next object from the results queue

- `double Bound() const`: returns a lower bound on the distance between the query predicate and the next object (without fetching any node from the queue)

Other methods:

- `~MTcursor()`: destructor
- `void FetchNode()`: fetches next node from the queue

This class implements the sorted access for the M-tree index structure. The definition of the class is given in the file `MTcursor.h`, and its implementation is included in the file `MTcursor.cpp`.

The user can begin a new sorted access to a tree with respect to a `MTpred` (see Section 3.8) by creating an instance of the `MTcursor` class. Then, it is sufficient to repeatedly send the `Next()` message to the so-created `MTcursor` object to retrieve, one-by-one, all the indexed `Object`s sorted in increasing order of distance with respect to the query predicate. The result of the `Next()` method is an `MTentry` whose `Object` coincides with the indexed object and whose `maxradius` represents the distance to the query predicate. Once the user is satisfied with the result, he/she can interrupt the sorted access by destroying the `MTcursor` object. The classical way to perform a sorted access is the following:

```
MT *tree=new MT;
tree->Open(...); // tree construction
...
MTpred pred(...); // query construction
MTcursor cursor(*tree, pred);
while(...)  { // continue until satisfied
    MTentry *e=cursor.Next();
    Object o=e->object(); // result object
    double dist=e->maxradius(); // distance from query predicate
    ...  // result evaluation
    delete e;
}
```

4 M-tree Files Overview

The M-tree software library is shipped as a Windows .zip file containing the following files:

filename	description
`MTree.ps`	This file
`GiSTdefs.h`	Redefinition of some GiST constants (should overwrite the GiST file)
`list.h`	Definition of an ordered list container
`Makefile`	A sample Makefile to compile the project under Solaris
`MT.h`	Global declarations and definition of the `MT` class
`MT.cpp`	Implementation of the `MT` class
`BulkLoad.cpp`	Implementation of the bulk-loading algorithm
`MTentry.h`	Definition of the `MTkey`, `MTpenalty`, and `MTentry` classes
`MTentry.cpp`	Implementation of the `MTentry` class
`MTfile.h`	Definition of the `MTfile` class
`MTfile.cpp`	Implementation of the `MTfile` class
`MTnode.h`	Definition of the `MTnode` class
`MTnode.cpp`	Implementation of the `MTnode` class
`MTobject.h`	Definition of the `Object` class
`MTobject.cpp`	Implementation of the `Object` class
`MTpredicate.h`	Definition and implementation of the `MTpred`, `Pred`, `AndPred`, `OrPred`, `MTquery`, and `TopQuery` classes, definition of the `Simplequery` class
`MTpredicate.cpp`	Implementation of the `SimpleQuery` class
`MTcursor.h`	Definition of the `MTcursor` class
`MTcursor.cpp`	Implementation of the `MTcursor` class
`Main.cpp`	A sample main() to show the use of M-tree features

The files `MTobject.h` and `MTobject.cpp` contain the definition of two simple `Object` classes: An object composed by a n-dimensional vector and an object composed by a list of weights (useful, as an example, to represent documents in the Vector Space Model for Information Retrieval applications [Sal88]).[2]

5 Copyright and Disclaimer

[2]Note that, for the latter application, `Objects` have different sizes (see Section 3.2).

merchantability, fitness for a particular purpose, or non-infringement. The authors shall not be liable for any damages suffered by licensee as a result of using, modifying or distributing this software or its derivatives.

References

[CA97] Weimin Chen and Karl Aberer. Efficient querying on genomic databases by using metric space indexing techniques. In *1st International Workshop on Query Processing and Multimedia Issues in Distributed Systems (PMIDS'97)*, Toulouse, France, September 1997.

[CP98] Paolo Ciaccia and Marco Patella. Bulk loading the M-tree. In *Proceedings of the 9th Australasian Database Conference (ADC'98)*, pages 15–26, Perth, Australia, February 1998.

[CPZ97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 426–435, Athens, Greece, August 1997.

[CPZ98] Paolo Ciaccia, Marco Patella, and Pavel Zezula. Processing complex similarity queries with distance-based access methods. In *Proceedings of the 6th International Conference on Extending Database Technology (EDBT'98)*, pages 9–23, Valencia, Spain, March 1998.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 562–573, Zurich, Switzerland, September 1995. `http://gist.cs.berkeley.edu:8000/gist/`.

[Pat99] Marco Patella. *Similarity Search in Multimedia Databases*. PhD thesis, Dipartimento di Elettronica Informatica e Sistemistica, Università degli Studi di Bologna, Bologna, Italy, 1999. `http://www-db.deis.unibo.it/~patella/`.

[Sal88] Gerard Salton. *Automatic Text Processing: The Transformational, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, Reading, MA, 1988.