Metric Spaces Library

www.sisap.org

Karina Figueroa[1,2], Gonzalo Navarro[2], Edgar Chávez[1]

[1] Escuela de Ciencias Físico-Matemáticas, Universidad Michoacana, Mexico
[2] Center for Web Research, Department of Computer Science, University of Chile

Bug reports, comments, and contributions to karina@fismat.umich.mx

November 14, 2008

We describe a library to support similarity searching in metric spaces. It contains various metric space and index implementations, as well as some tools to evaluate their performance for similarity searching. The library is is an integral part of the new conference *Similarity Search and Applications (SISAP)* created in 2008. It was defined and initially populated by the authors, but we expect it to grow with other contributions over time.

The goal of similarity searching is, given a finite set of objects $\mathbb{U}$ (called a *database*) drawn from a (possibly infinite) universe $\mathbb{X}$, and a *distance function* $d(\cdot,\cdot)$ defined among objects of $\mathbb{X}$, preprocess $\mathbb{U}$ and build a data structure (called an *index*) so that similarity searches can be carried out on that set. The objects are seen as black boxes, on which the only operation one can perform is to compute distances, and nothing else should be assumed on them. As the distance function is in many cases expensive to compute, it is customary to take the number of distance computations as the complexity measure for index construction and searching.

In particular, for *metric space* searching, the distance d is assumed to be strictly positive (except $d(x,x) = 0$, which must always hold), symmetric ($d(x,y) = d(y,x)$), and to satisfy the *triangle inequality*, that is, $d(x,z) \leq d(x,y)+d(y,z)$. The library focuses on the two most popular types of queries:

range queries (given an object $q \in \mathbb{X}$, present or not in the database $\mathbb{U}$, and a radius r, retrieve all database objects within distance r to q), and *kNN or k-nearest-neighbor queries* (given an object $q \in \mathbb{X}$ and an integer K, retrieve the K database objects closest to q, breaking ties arbitrarily).

It is always possible to answer any similarity query by a simple brute-force scanning of the database, computing the distances between the query and each database object. Yet, this is might be too slow for many applications. The performance of the indexes is measured against the brute-force approach. There are indexes that perform better or worse depending on the space, and there are also intrinsically easier and harder metric spaces. In hard metric spaces (where usually the histogram of distances is highly concentrated) even the strongest indexes fail to perform much better than a linear scan. Hence the interest in considering different index and metric space combinations. It is not our intention here to demonstrate the relevance of the area; for this sake we refer the reader to some relevant surveys and books [7, 9, 19, 15].

The library is designed for data sets that are small enough to let the index fit in main memory, and static enough for a whole reconstruction being a decent alternative upon database changes. The programmed indexes, although they are reasonable implementations, are not particularly optimized for speed. Yet, they are correct implementations of data structures proposed in the literature, in terms of number of distance computations performed. In addition, we have considered only exact algorithms, that is, those always giving correct answers. As seen, we have left aside several possible extensions and optimizations in our aim to have a concrete and reasonably complete library of the existing basic work in the area. The site will contain a section for other implementations which either do not fit in this restricted setup, or we have just not had time yet to integrate into the library. It is possible to give more structure and standardization in the future to some of those extensions to the basic problem.

Section 1 of this manual describes the essentials of the library organization and the basic instructions to make it work. Section 2 describes the indexes, metric spaces, and databases that currently populate the library. Finally, Section 3 gives the more advanced information necessary to enrich it with new indexes, metric spaces, or databases.

The library is in C language. We might port it to C++ in the future.

1 Library Organization

The library can be downloaded from `http://www.sisap.org`. It consists of the following major files and directories.

- `Readme.txt`: A pointer to this manual.
- `Copyright.txt`: Copyright information. The code is licensed under a GNU Public License.
- `src`: Source codes for indexes (subdirectory `indexes`), metric spaces (subdirectory `spaces`), and other programs.
- `dbs`: Databases for the different metric spaces.
- `lib`: Where the compiled modules are stored.
- `bin`: Where the compiled executables are stored.
- `doc`: This documentation.
- `others`: Other contributions not (yet) normalized into the general framework.

Almost all the file structure is obtained by expanding the main gzipped tarball in the site. Because the files under `dbs/` are very large, they must be downloaded separately from subdirectory `dbs/` in the site. Apart from a gzipped tar containing them all, each database under each space can be downloaded individually from a gzipped tarball under the corresponding subdirectory `dbs/SPACE`. When expanded it will create the corresponding subdirectory `dbs/SPACE/DATABASE` and the files below it.

1.1 Library Architecture

There are four main actors in the library: *metric spaces*, *indexes*, *databases*, and *programs.*

Metric spaces are descriptions of metric object data types. A metric space provides essentially a (usually opaque) data type plus a distance function among pairs of such objects, but it also implements functions to load/save objects from/to disk, etc. Implementations are found within subdirectory `src/spaces`. The distance function returns a `float` or `int` type depending on whether it is of continuous or discrete nature, respectively.

Indexes implement data structures for indexing metric spaces, and should work with any of them. An index implements functions to build the index from a database, run range and kNN queries, etc. They are found within subdirectory `src/indexes`.

Databases are instances of a given metric space. They can be either randomly generated (e.g. uniformly distributed unitary cubes) or gathered from some public repository. They are files stored in the correct format so that they can be read with the corresponding metric space implementation. The databases for the metric space `SPACE` are stored under directory `src/dbs/SPACE`, whereas the generators are in `src/spaces/SPACE`.

Programs use a combination of an index, a metric space, and a specific database for that metric space, to carry out some task. We provide implementations of basic programs that should be useful for carrying out experiments, and those serve as examples for users of the site that wish to use metric space implementations for other specific purposes. The programs we provide are `src/build.c` and `src/query.c`. The first builds and stores an index structure for a given database; the second takes the index and runs queries on it.

1.2 Compiling and Running

Compiling the library requires an ANSI standard C compiler. The library has been compiled successfully on Unix, Linux and MacOS systems.

After downloading the sources, change to the library root directory (from which directories such as `bin`, `doc`, and `src` branch off), and enter the command `make`. This will create, under `bin`, executables for any combination of program, index, and metric space available in the library. (`make clean` will remove all compiled modules and executables in `bin` and `lib`).

More precisely, for any index implemented under directory `src/indexes/INDEX` and metric space implemented under directory `src/spaces/SPACE`, this command will create an executable called `bin/build-INDEX-SPACE` and bin/query-INDEX-SPACE. One can also run make using a particular combination as an argument in order to build just that combination.

Once the executables are built, the programs are used according to the syntax described next.

1.2.1 Build

As explained, build-*-* commands take a particular database and create an index data structure for them. They are used as follows:

```
build-INDEX-SPACE <database name> <size> <index name>
                  <construction parameters>
```

where <database name> is the name of the database of objects (usually a file or directory name, or a set of files named <database name>.*, depending on how the metric space stores databases, see Section 2.2); <index file> is the name of the index that will be generated (usually a file or directory name, or a set of files named <index name>.*); and finally <construction parameters> are specific parameters for the index data structure INDEX. Those parameters depend on the specific index and adhere to its particular syntax (see Section 2.1). Finally, <size> is a nonnegative number. If it is zero, all the database will be indexed; otherwise only the first <size> elements will be considered for the index.

Example After creating a database file euclidean.10.20000 containing 20,000 uniformly-generated points in a 10-dimensional unit cube under Euclidean distance, one might run

```
build-pivots-vectors euclidean.10.20000 0 myindex 6
```

which will generate a pivot-based index for those 20,000 points (the zero means that the index should be build for the whole database) and write that index in file myindex. The final number, 6, is a parameter for the pivots index and means that it will use 6 pivots to index the database.

1.2.2 Query

Commands query-*-* take an index built with build-*-* and then answer queries from the standard input until a terminator query is read. They also print (to the standard error output) statistics on the performance of the searches (average number of distance computations, and CPU/system times).

The entries (which of course can be redirected from a file created beforehand) are given as a sequence of lines of the form `<k>,<object>`. Parameter `k` can be preceded by a minus sign or not. In not preceded by a minus sign, then `k` represents a (real or integer) distance and denotes a range query with radius `k`. If preceded by a minus sign (`k` is `-`K), it represents a kNN query retrieving the K nearest neighbors. The sequence of queries terminates with a line of the form `-0`.

The query `<object>` is an object of the metric space for which the index has been created, belonging or not to the database. It is written in a syntax that is defined by the appropriate metric space implementation, which includes rules for parsing a sequence of symbols into an object (any character up to the end of line is taken as part of the object representation). Some metric spaces (e.g. strings, vectors) might permit a direct representation of objects, while others (e.g. images, documents) might use identifiers into a database.

The output of the query is given in the form of a sequence of lines, with the representation of the objects found (this representation is also defined by the metric space implementation, see Section 2.2). This can be redirected to `/dev/null` if one is interested only in the statistics.

Example After creating `myindex` as in the previous example, we can now run

```
query-pivots-vectors myindex
```

which will load `myindex` into main memory and wait for queries in the standard input. We can then write, say,

```
0.7,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5
-3,1,1,1,1,1,1,1,1,1,1
-0
```

where the first query asks for all objects in `euclidean.10.20000` that are within Euclidean distance 0.7 of the point $(0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5)$; the second asks for the 3 nearest neighbors of the point $(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$; and the third terminates the program.

1.2.3 Creating Databases and Query Sets

Databases are stored within subdirectories `dbs/SPACE`, whereas programs to generate synthetic databases, as well as query sets from databases (i.e., inputs to `query-*-SPACE` programs), are stored in `src/spaces/SPACE`. The library contains at least one database and/or random generator per metric space. See Section 2.2.

2 Current Library Contents

2.1 Indexes

The following indexes are currently implemented in the library. Their name is also the subdirectory where they can be found from `src/indexes`. We specify, in each case, the construction parameters that must be passed on to `build-*-*` programs. Our aim is not to explain each index; references to the literature are given for this sake. In most cases the publications describe the structure and the range searching method; the kNN method is more variable. We indicate the kNN method implemented in each case, trying to follow that of the original publication when it exists.

aesa (Approximating Eliminating Search Algorithm [17].) It requires quadratic space to store all inter-object distances in the database, yet it usually performs the lowest number of distance computations at query time. It has no construction parameters. The kNN method is as described in [17].

iaesa (improved AESA [8].) A version of AESA that changes the way to choose the next pivot. The new predictor is based on a k-length permutation. Its construction parameter is k. The kNN method is as in AESA.

pivots (Linear AESA [12].) is a table storing, for a database of n objects, kn distances, where k is a parameter. We implement a simplified variant where the query is always compared against all the k pivots (the real LAESA is more sophisticated). There should be no real difference except for large k. The parameter of the algorithm is therefore k. The kNN method sorts the database by sum of distances to the pivots and scans it in that order, in the same spirit of AESA.

bkt (Burkhard-Keller Tree [5].) It takes linear space and was originally defined only for discrete distances. However we generalized it to continuous distances by using fixed-width rings. It has construction parameters b and s. The first is the bucket size in the leaves, the second is the ring width. The kNN method explores rings whose distance to the center is less and less similar with the distance between the query and the center.

fqt (Fixed-Queries Tree [1].) A BKT variant using a single pivot per level. The construction parameters are b and a, where b is as for BKTs and a is the tree arity (the fixed step is computed considering minimum and maximum distances to the level pivot). The kNN method is as for the BKT.

fqh (Fixed-Height Queries Tree [1].) A FQT variant using exactly h levels, and thus spending between n and nh integers of space. The construction parameters are h and a, where a is as for FQTs. The kNN method is as for the BKT.

ght Generalized-Hyperplane Tree [16] and Bisector Tree [10].) These are trees requiring linear space. As the structures are identical and their query strategies are rather similar, we implemented a version that gets the best from both (technically, we prune by hyperplane and by covering radius criteria, see [7]). We also generalize the tree to be a-ary, inducing a Voronoi-like partitioning at each node (in the original structures $a = 2$). The parameter for the structure is thus a. Note that this is not yet a GNAT [4], whose pruning criteria are stricter (and take more space). The kNN method enters recursively into the subtrees, choosing the order according to closeness to the centers.

mvp (Vantage Point Tree [18] and Multi Vantage-Point Tree [3].) The first is a balanced binary tree requiring linear space. The second extends it to arity a and to multiple pivots per node. Rings are defined using percentiles. We only implemented the extension to multi-ary, yet each node still has just one pivot. Its parameters are b and a, the first being the bucket size at the leaves. The kNN method is as for the BKT.

sat (Spatial Approximation Tree [13].) It induces a self-adjusting Voronoi-like partitioning of the space, takes linear space, and has no construction parameters. The kNN method is the range-optimal one described in [13].

lcluster (List of Clusters [6].) It is a list of balls taking linear space, and has construction parameter b (number of elements in each ball). The smaller b, the slowest to build. The kNN method scans the balls in sequence, entering into each one that cannot be discarded with the current kNN candidates.

2.2 Metric Spaces and Databases

As explained, each metric space might have different databases conforming with it. Again, the name of the metric space corresponds to the name of the directory under `src/spaces`, where the code resides, and the directory under `dbs`, where the databases reside.

2.2.1 Vectors (directory `vectors`)

Definition. A vector in dimension d is a d-tuple of real numbers. There are different distances that can be used between two vectors. A popular family is the Minkowski distances, or L_p norms:

$$L_p((x_1,x_2,\ldots,x_d),(y_1,y_2,\ldots,y_d)) \quad = \quad \left(\sum_{1\le i\le d} |x_i-y_i|^p\right)^{1/p}$$

for any $p \geq 1$, and including the limit case

$$L_\infty((x_1,x_2,\ldots,x_d),(y_1,y_2,\ldots,y_d)) \quad = \quad \max_{1\le i\le d} |x_i-y_i|$$

The most popular variants are L_1 (or Manhattan distance), L_2 (Euclidean distance), and L_∞ (maximum distance). To factor out code, we will treat these three distances as a single metric space, and let each database specify which distance it uses (note this is not strictly correct, but raises no problems).

Format. A database is stored as a file, with some header information and then the vector coordinates. The header information consists of a first line with three space-separated integers: the dimension, the number of vectors, and the type of distance (`1`, `2`, or `i`). Then the vectors are given as their comma-separated coordinates, one vector per line.

To avoid problems of float representations across machines, the next spaces and generators store/produce ASCII representations of the vectors. To render them usable as a metric space, run `make` in `src/spaces/vectors` and then run

```
convertcoords <ascii file> <binary file>
```

where `<ascii file>` is the file where the vectors are stored in ASCII format, and `<binary file>` is a file that will be created with the vectors rewritten in usable format.

Databases. The following databases and generators are currently present in the library, the databases under directory `dbs/vectors` and the generators under directory `src/spaces/vectors`. The names of the spaces match their subdirectories.

The seed value (for random number) used by these generators can be controled using an environment variable SISAP_SEED. If the variable is not defined then the seed is initialized as a random value, such as the system time.

uniform: A program that generates points in the cube $[0,1)^d$ with uniform coordinate distribution, for any metric. It is built using `make` on that directory, and then invoked as

```
gencoords <l> <dim> <number> <file>
```

which will generate in `<file>` a database of `<number>` points in `<dim>` dimensions, and set the distance to L_1 if `<l>` is `1`, to L_2 if `<l>` is `2`, and to L_∞ if `<l>` is `i`. As explained, the file will be generated in ASCII format and then needs to be converted.

gaussian: A program that generates clustered sets of points with Gaussian distribution. It is based on *gaussora*, available at the 6th DIMACS Implementation Challenge (`http://dimacs.rutgers.edu/Challenges/Sixth/software.html`. After running `make` to compile, execute:

```
gengauss <num clusters> <dim> <var> <num points>
```

which will generate `<num points>` points in `<dim>` dimensions, clustered around `<num clusters>` centers. The coordinates of the centers are uniformly distributed in $[0, 1)$. The variance of the Gaussians around the points is `<var>` and points are assigned to the clusters with uniform probability. The standard output must be redirected to a file where the ASCII version of the resulting vector space instance will reside.

nasa: a set of 40,150 20-dimensional feature vectors, generated from images downloaded from NASA (at `http://www.dimacs.rutgers.edu/Challenges/Sixth/software.html`) and with duplicate vectors eliminated. The Euclidean distance is used.

colors: a set of 112,544 color histograms (112-dimensional vectors) from an image database (at `http://www.dbs.informatik.uni-muenchen.de/~seidl/DATA/histo112.112682.gz`, there are others in the same page) with duplicates removed. Any quadratic form can be used as a distance. We chose Euclidean distance as the simplest meaningful alternative.

Queries. For input/output, vectors are represented as a comma-separated list of their components: `x1,x2,...,xd`. Queries in `query-*-vectors` expect the objects in this format, and results are printed in this format.

When massive experiments are run, it is convenient to have a way to automatically generate input files for `query-*-vectors`. We provide a tool in directory `src/spaces/vectors`, built with `make`, and invoked as

```
genqueries <database file> <from> <num queries> <k> <perturb>
```

which will send `<num queries>` to the standard output, all their `k` values being precisely `<k>` (recall that it is a range query if positive and a kNN query if negative), except for a terminating `-0`. The points in the queries will be randomly chosen from `<database file>` (starting at the (`<from>`$+1$)-th element), and one random coordinate will be perturbed by $\pm$`<perturb>`.

This program can be used in several ways: (1) run it on the same database that will be queried, with zero perturbation, so the queries will be database elements; (2) run it on the same database that will be queried, with nonzero perturbation p, so the queries will be at distance p (in any Minkowski metric) to some database element, and thus querying with radius p will retrieve at least one element; (3) run it on the same database that will be queried, but having built it with the first `<from>` elements and now choosing the queries from the other elements; (4) run it on a different database, with zero perturbation, so the queries will be independent of the database elements (e.g. generated with the same or another distribution).

2.2.2 Strings (directory `strings`)

Definition. There are several algorithms to define a distance between two strings of characters. A popular one is the *edit distance* or *Levenshtein distance* [11], which is the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character. The distance is obviously integer-valued. A dynamic programming formula to compute it follows (for strings s_1 and s_2; s_ic_j represents the string s_i concatenated with character c_j:

$$
\begin{aligned}
d(s_1, s_2) &= |s_1| \text{ if } s_2 \text{ is empty} \\
d(s_1, s_2) &= |s_2| \text{ if } s_1 \text{ is empty} \\
d(s_1c_1, s_2c_2) &= d(s_1, s_2) \text{ if } c_1 = c_2 \\
d(s_1c_1, s_2c_2) &= 1 + \min(d(s_1, s_2), d(s_1, s_2c_2), d(s_1c_1, s_2)) \text{ if } c_1 \neq c_2
\end{aligned}
$$

Format. A database is stored as a file, with one string per line. It is recommended that the strings do not follow any particular order (e.g. lexicographical) so that it can be processed sequentially without producing any artifact. A simple way to achieve this is to run `sort -R <file> <new file>`, which randomly reorders de lines. Yet, option `-R` is not supported in all distributions. The databases included in the library are already disordered.

Databases. The library contains the following sets of strings and generators, the sets under `dbs/strings`, and under `src/spaces/strings` the generators.

dicts: Dictionaries in several languages, of unknown source. We disordered them to avoid dependences on lexicographical ordering.

vocabs: A program to process a collection of files in a directory in order to extract the corresponding vocabularies (set of distinct words). After running `make` in the directory, and keeping that as the current directory, the program is used as

```
genvocab <dir> <vocab file>
```

where `<dir>` is the directory where only the bare files to process should be included, and `<vocab file>` is the vocabulary file to generate. It must be possible to write in the directory where `genvocab` is executed, and a subdirectory `tmp` will be removed if it exists. Words are defined as maximal contiguos sequences of letters and digits, and are converted to lowercase.

After generating the vocabulary, it is recommended to randomly reorder it, as explained.

genes: A set of 20,660 DNA sequences of genes of *Listeria monocytogenes*, downloaded from `http://www.broad.mit.edu/annotation/genome/listeria_group/MultiDownloads.html`.

Queries. For input/output, strings are written in plain form, as a sequence of symbols, and terminated with a newline. Queries in `query-*-strings` expect the objects in this format, and results are printed in this format.

When massive experiments are run, it is convenient to have a way to automatically generate input files for `query-*-strings`. We provide a tool in directory `src/spaces/strings`, built with `make`, and invoked as

```
genqueries <database file> <from> <num queries> <k> <perturb>
```

which will send `<num queries>` to the standard output, all their `k` values being precisely `<k>` (recall that it is an integer-range query if positive and a kNN query if negative), except for a terminating `-0`. The strings in the queries will be randomly chosen from `<database file>`, and `<perturb>` random edit operations will be carried out on them (there is a slight chance that some edits cancel others, so that the final distance between the original and perturbed string could be less than `<perturb>`).

This program can be used in several ways: (1) run it on the same database that will be queried, with zero perturbation, so the queries will be database elements; (2) run it on the same database that will be queried, with nonzero perturbation p, so the queries will be at distance at most p to some database element, and thus querying with radius p will retrieve at least one element; (3) run it on the same database that will be queried, but having built it with the first `<from>` elements and now choosing the queries from the other elements; (4) run it on a different database, with zero perturbation, so the queries will be independent of the database elements.

2.2.3 Documents (directory `documents`)

Definition. The typical document similarity used under the vector space model [2] can be adapted to define a metric space. In this model each text document is seen as a vector in a high-dimensional space (one coordinate per vocabulary word), and the so-called *cosine similarity* is used, which measures the number and relevance of shared terms between two documents. The inverse of the cosine turns out to be a metric.

Let $t_1, \ldots, t_k$ be the set of terms (vocabulary words of the whole collection) and $d_1, \ldots, d_n$ the documents. A document d_i is modeled as a vector

$$\overrightarrow{d_i} = (w(t_1, d_i), \ldots, w(t_k, d_i))$$

where $w(t_r, d_i)$ is the *weight* of term t_r in document d_i. A popular formula to compute the weight is

$$w(t_r, d_i) = w_{r,i} = \frac{\phi_{r,i} \cdot \log \frac{n}{n_r}}{|\overrightarrow{d_i}|} = \frac{\phi_{r,i} \log \frac{n}{n_r}}{\sqrt{\sum_{s=1}^{k} \left(\phi_{s,i} \log \frac{n}{n_s}\right)^2}}$$

where $\phi_{r,i}$ (the term frequency) is the number of times term t_r appears in document d_i, and n_r is the number of documents where t_r appears. Finally, we define the distance function as the angle between their vectors:

$$d(d_i, d_j) = \arccos\,(\overrightarrow{d_i} \cdot \overrightarrow{d_j}) = \arccos\left(\sum_{r=1}^{k} w_{r,i} \cdot w_{r,j}\right).$$

Format. A database is stored as a directory, with one vector per file. To avoid problems of float representations, the next spaces and generators store/produce ASCII representations of the vectors. To render them usable as a metric space, you must run `make` in `src/spaces/documents` and then run

```
convertdir <ascii dir> <binary dir>
```

where `<ascii dir>` is the directory where the vectors are stored in ASCII format, and `<binary dir>` is a directory that will be created with the vectors rewritten in usable format.

Databases. We have the following document databases and generators in the library, the former under `dbs/documents` and the latter under `src/spaces/documents`.

short: A set of 25,960 short news articles extracted from Wall Street Journal 1987-1989 files from TREC-3 collection. Note that only the vector representations of the documents are maintained, not the original articles.

long: The same news articles, now taken in the original format where the set is divided into 1,265 files of approximately 1 MB each. We take each file as a document.

tfidf: A program to process a collection of files in a directory in order to extract the corresponding vectors and create a new database usable as an instance of this metric space. After running `make` in the directory, and keeping that as the current directory, the program is used as

```
index <dir> <target dir>
```

where `<dir>` is the directory where only the files to index should be included. It will create `<target dir>` and then, for each file under `<dir>`, a file of the same name will be created under `<target dir>` (might take some time). This directory `<target dir>` is the new database, still in ASCII format. It must be possible to write in the directory where `index` is executed, and a subdirectory `tmp` will be removed if it exists. Words are defined as maximal contiguos sequences of letters and digits, and are converted to lowercase.

Queries. Documents are complex objects and thus no simple representation of elements of the universe is implemented. Instead, the database is identified with the universe and objects are input as their integer identifier in the database (between 1 and n, where n is the database size). For output purposes we use the name of the corresponding file, as a courtesy to the final user.

No perturbation mechanism is provided either, thus queries will always be database elements. It is not possible either to run queries from another database, as the vectors are not comparable. To provide a mechanism to have queries not belonging to the database, we can use the mechanism to build the database with the first $n' < n$ elements and pick queries from the last $n - n'$ elements. For this sake we provide the program (built with `make` in directory `src/spaces/documents`)

```
genqueries <database dir> <from> <num queries> <k>
```

which will send `<num queries>` to the standard output, all their `k` values being precisely `<k>` (recall that it is a range query if positive and a kNN query if negative), except for a terminating `-0`. The document identifiers (i.e. integers) in the queries will be randomly chosen between `<from>` and n, being n the database size.

2.2.4 Face Images (directory `faces`)

Definition. *Faces* are vectors of featured extracted from a real image. The usual distance to compare faces is the Euclidean distance.

One of the most popular database of face images is FERET [14]. FERET was designed to develop automatic face recognition capabilities that could be employed to assist security, intelligence and law enforcement personnel in the performance of their duties.
`http://www.itl.nist.gov/iad/humanid/feret/feret_master.html`

Format. The database is stored as a file, with some header information followed by feature vectors and an identifier. The header consists of a first line with 2 space-separed integers: the number of vectors and components. Then the vectors are given as their comma-separed coordinates and its identifier, one face image per line. To avoid problems with the representation of floating point numbers across machines, this data is stored in plain ASCII.

Databases. We have two face images databases in the library. One in the directory `dbs/faces` and one in `src/spaces/faces`.

- train-762_761: It is a set of 762 faces with 761 components. It is used for training.
- test-254_761: It is a set of 254 faces with 761 components. It is used for testing.

Queries For input/output, face images are written in plain form as a comma-separed list of features: $x_1, x_2, \ldots, x_d$ and then an identifier. We provided a set of 254 queries and a tool to add the arguments of the query (*knn* or range query).

```
genqueries <database file> <from> <num queries> <k>
```

which will send `<num queries>` to standard output, all `k` values being precisely `<k>` (recall that it is a range query if positive and a *knn* query if negative), except for a terminating `-0`. The points in the queries will be randomly chosen from `<database file>` (starting at the (`<from>` $+ 1$)-th element).

3 Contributing to the Library

It is possible to add new indexes, metric spaces, and databases to the library. There is also a directory `others` to add contributions that do not (yet) fit within the general framework.

The information to add a database is already given in the description of the metric spaces (Section 2.2). Thus we focus on adding new metric spaces and new indexes.

As for new programs using those indexes, you can take `src/build.c` and `src/query.c` as examples. Bear in mind that, in order to make adding spaces and indexes as easy as possible, we are requiring just the most basic functionality from them.

3.1 Metric Spaces

To add a novel metric space `SPACE`, `src/spaces/SPACE` and `dbs/SPACE` directories must be created. In the second one must create at least one database, as explained. In the first directory one must actually implement the metric space, that is, provide the source code (with GNU License) that computes the distance, manipulates databases, and so on.

The model is as simple as possible: one *opens* a database (of size n), and from then on database objects can be referred by their identifier (an integer between 1 and n; the `int` type is thus also called `Obj`). Integer 0 (`NewObj`) is reserved to create a new object not belonging to the database (usually a query) and -1 (`NullObj`) to denote a null object. At most one database can be open at any time. An index, for example, can store those integer identifiers in its structures, and those will be meaningful again once the database is opened.

Concretely, the source file must implement the interface `src/obj.h`, which includes the following functions.

`int openDB (char *descr)` loads database called `name`. If there is another database already open, it is closed first. The function returns the number of objects in the database. It must guarantee that object identifiers are the same upon different openings of the same database.

`void closeDB (void)` closes the currently open database.

`Tdist distanceInter (Obj o1, Obj o2)` computes the distance between objects `o1` and `o2`. The return type is `int` or `float` depending on the compilation option (see later). Note that this is invoked from outside as `distance (o1,o2)`, which is a macro that in addition counts the number distance evaluations in variable `numDistances`.

`Obj parseobj (char *str)` parses description `str` into an object identifier that is returned. The identifier must be `NewObj` if the object does not belong to the DB, and the module must ensure that operations like `distance(NewObj,o2)` work properly after that.

`void printobj (Obj obj)` prints a user-friendly description of object identifier `Obj` (or the identifier itself if the objects are too complex to be printed). A newline is printed at the end.

Once the code is complete, add a line in `src/Makefile` (not before the line starting with `all:`) as follows:

```
lib/space-SPACE.o: src/spaces/SPACE/FILE.c
        gcc $(CFLAGS) -o lib/space-SPACE.o \
                      -c src/spaces/SPACE/FILE.c
```

if your distance is continuous (`float` type), or

```
lib/space-SPACE.o: src/spaces/SPACE/FILE.c
        gcc $(DFLAGS) -o lib/space-SPACE.o \
                      -c src/spaces/SPACE/FILE.c
```

otherwise, where SPACE is your new directory and FILE is your new .c file implementing obj.h. Adapt this basic rule to your needs if more source files must be considered. Extra compiled modules (.o) are better stored within lib/spaces/SPACE to avoid name clashes with unknown present or future spaces.

Once lib/space-SPACE.o is built, then for each index INDEX that you wish to combine with your new space SPACE, add a line as follows:

```
bin/build-INDEX-SPACE: lib/buildC.o \
                       lib/indexC-INDEX.o lib/space-SPACE.o
        gcc $(FLAGS) -o bin/build-INDEX-SPACE lib/buildC.o \
                     lib/indexC-INDEX.o lib/space-SPACE.o ETC
bin/query-INDEX-SPACE: lib/queryC.o \
                       lib/indexC-INDEX.o lib/space-SPACE.o
        gcc $(FLAGS) -o bin/query-INDEX-SPACE lib/queryC.o \
                     lib/indexC-INDEX.o lib/space-SPACE.o ETC
```

if your distance is continuous, or

```
bin/build-INDEX-SPACE: lib/buildD.o \
                       lib/indexD-INDEX.o lib/space-SPACE.o
        gcc $(FLAGS) -o bin/build-INDEX-SPACE lib/buildD.o \
                     lib/indexD-INDEX.o lib/space-SPACE.o ETC
bin/query-INDEX-SPACE: lib/queryD.o \
                       lib/indexD-INDEX.o lib/space-SPACE.o
        gcc $(FLAGS) -o bin/query-INDEX-SPACE lib/queryD.o \
                     lib/indexD-INDEX.o lib/space-SPACE.o ETC
```

if it is discrete. Here ETC stands for any other library you need to link to make INDEX or SPACE work. Remember also to add the new *-*-SPACE targets to the main target all: in the beginning of the makefile.

Finally, it is strongly advised that new metric spaces include mechanisms to generate random queries, as flexible as possible given the space. The implemented spaces can serve as inspiration.

3.2 Indexes

Unlike metric spaces, indexes should work both for continuous and discrete distances. Thus you should use Tdist as the distance data type, and it will become float when you compile using -DCONT and int when you compile using `-DDISCR`. The index will be compiled in both flavors to combine the `.o` files with different spaces.

To include a new index in the library you should create a new directory `src/indexes/INDEX` and include your source code (with GNU License) within it. An index must implement the interface given in `src/index.h`:

`Index build (char *dbname, int n, int *argc, char ***argv)` builds an index (and returns a handle of type `Index`, which is `void*`) for the currently open database (named `dbname`), considering only the first `n` elements. Construction parameters come directly from the command line, `*argc` and `*argv`, which must be consumed consistently.

`void freeIndex (Index S, bool closedb)` frees the index and, if `closedb`, also closes the database.

`void saveIndex (Index S, char *fname)` stores the index in filename (or directory, or set of files) `fname`. Objects are stored as their integer identifiers. The index must store the database name, as it will not be supplied at loading time.

`Index loadIndex (char *fname)` loads the index from `fname`, also opening the database.

`int search (Index S, Obj obj, Tdist r, bool show)` is a range search for query `obj` with radius `r` in index `S`, returning the number of database elements found. Those are printed to the stdout if `show`.

`Tdist searchNN (Index S, Obj obj, int k, bool show)` is a kNN search for the `k` nearest neighbors of query `obj` in index `S`, returning the distance to the farthest answer. Those are printed to the stdout if `show`.

Once the code is complete, add two lines in `src/Makefile` (not before the line starting with `all:`) as follows:

```
lib/indexC-INDEX.o: src/indexes/INDEX/FILE.c
        gcc $(CFLAGS) -o lib/indexC-INDEX.o \
                      -c src/indexes/INDEX/FILE.c
lib/indexD-INDEX.o: src/indexes/INDEX/FILE.c
        gcc $(DFLAGS) -o lib/indexD-INDEX.o \
                      -c src/indexes/INDEX/FILE.c
```

where INDEX is your new directory and FILE is your new .c file implementing index.h. Again, this can be adapted to the case where more sources/modules are needed. Extra compiled modules (.o) are better stored within lib/indexes/INDEX to avoid name clashes with unknown present or future indexes.

Then, for each space SPACE that you wish to combine with your new index INDEX, add a line as follows:

```
bin/build-INDEX-SPACE: lib/buildC.o \
                       lib/indexC-INDEX.o lib/space-SPACE.o
        gcc $(FLAGS) -o bin/build-INDEX-SPACE lib/buildC.o \
                     lib/indexC-INDEX.o lib/space-SPACE.o ETC
bin/query-INDEX-SPACE: lib/queryC.o \
                       lib/indexC-INDEX.o lib/space-SPACE.o
        gcc $(FLAGS) -o bin/query-INDEX-SPACE lib/queryC.o \
                     lib/indexC-INDEX.o lib/space-SPACE.o ETC
```

if the distance of the space is continuous, or

```
bin/build-INDEX-SPACE: lib/buildD.o \
                       lib/indexD-INDEX.o lib/space-SPACE.o
        gcc $(FLAGS) -o bin/build-INDEX-SPACE lib/buildD.o \
                     lib/indexD-INDEX.o lib/space-SPACE.o ETC
bin/query-INDEX-SPACE: lib/queryD.o \
                       lib/indexD-INDEX.o lib/space-SPACE.o
        gcc $(FLAGS) -o bin/query-INDEX-SPACE lib/queryD.o \
                     lib/indexD-INDEX.o lib/space-SPACE.o ETC
```

if it is discrete. Here ETC stands for any other library you need to link to make INDEX or SPACE work. Remember also to add the new *-INDEX-* targets to the main target all: in the beginning of the makefile.

Other potentially useful sources can be found directly in src, for example generic code to handle the priority queue needed for range-optimal kNN searching, bucket management, etc.

References

[1] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proceedings 5th Combinatorial Pattern Matching (CPM'94)*, volume 807 of *Lecture Notes in Computer Science*, pages 198–212, 1994.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[3] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 357–368, 1997. Sigmod Record 26(2).

[4] S. Brin. Near neighbor search in large metric spaces. In *Proceedings 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.

[5] W. Burkhard and R. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, April 1973.

[6] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[7] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[8] K. Figueroa, E. Chávez, G. Navarro, and R. Paredes. On the lowest cost for proximity searching in metric spaces. *5th International Workshop on Experimental Algorithms*, 4007:270–290, May 2006.

[9] G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions Database Systems*, 28(4):517–580, 2003.

[10] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, 9(5), 1983.

[11] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics - Doklady*, volume 10, pages 707–710, 1966.

[12] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.

[13] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[14] P. Phillips, H. Wechsler, J. Huang, and P. Rauss. The FERET database and evaluation procedure for face recognition algorithms. *Image and Vision Computing Journal*, 16(5):295–306, 1998.

[15] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[16] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[17] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[18] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings 4th ACM-SIAM Symposium on Discrete Algorithms (SODA'93)*, pages 311–321, 1993.

[19] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems.* Springer, 2006.