National and Kapodistrian University of Athens
Department of Informatics and Telecommunications

K23α: Software Development for Information Systems

Exercise #1

Winter Semester 2024-2025

Team Members	**Student Numbers**
Michael-Raphael Kostagiannis	1115202100078
Giorgos Sofronas	1115202100180
Styliani Tziouma	1115202100184

Introduction

All the project requirements have been fulfilled. The main program has been thoroughly tested with various arguments. Neither the main program, nor the unit tests have any memory leaks. The project's workload was equally distributed amongst our team members. Specifically, here is a comprehensive list of each member's work:

- Michael:
  - Directed Graph: constructor, destructor, insert
  - Robust Prune
  - main: parse_parameters
- Giorgos:
  - Directed Graph: remove, get_neighbors
  - Vamana (medoid, vamana implementation, read/write graph from/to file)
  - Makefile structure
- Styliani:
  - Vectors class
  - Greedy Search
  - main: menus

Compilation and Execution

Along with the source and header files, found under `src/` and `include/` respectively, the project is instrumented using a group of three Makefiles:

1. Present under the `src/` directory, responsible for the compilation of the main executable.
2. Can be found under the `tests/` directory. It is responsible for the compilation of the unit tests.
3. At the root of the project, from which both aforementioned makefiles are being controlled.

As far as users are concerned, **they should only use the makefile at the root of the project** to compile the main program and/or the unit tests.
The available commands are listed in the following table:

Compilation of the main program:	`make main`
Compilation of the unit tests:	`make tests`
Compilation of both:	`make`
Cleanup command:	`make clean`

Object files are stored inside a real-time created directory called `build/`.

The program is executed using the following command:
`./k23a -b <base file> -q <query file> -g <groundtruth file> -t <field type> -n <num base vectors> -m <num queries> -d <dimension> -k <neighbors> -a <alpha> -l <L> -r <R>`
where each flag indicates the following:

- -b: The base vectors file.
- -q: The query vectors file.
- -g: The ground truth vectors file.
- -t: The field type of the vectors' coordinates. If 0 is given, vectors consist of unsigned chars. If 1 is given, vectors consist of floats.
- -n: The number of vectors found in the base vectors file.
- -m: The number of vectors found in the query vectors file.
- -d: The dimension of all vectors.
- -k: The number of the nearest neighbors.
- -a: The distance threshold used by Vamana. Must be greater than 1.
- -l: Search list size used by Greedy Search. Must be greater or equal to k.
- -r: The degree bound used by Robust Prune.

Parameters can be given in any order. Here is a sample execution:

```
./k23a -b siftsmall/siftsmall_base.fvecs -q siftsmall/siftsmall_query.fvecs
-g siftsmall/siftsmall_groundtruth.ivecs -t 1 -n 10000 -m 100 -d 128
-k 100 -a 1.1 -l 150 -r 100
```

The specific arguments produce astonishing results, with an average **recall of 99.8%** and total execution time of $\sim$ **55 seconds**.

Unit Testing

All of our functions and methods have been extensively tested, using acutest: a simple header-only C/C++ unit testing facility. The tests are focused on checking a wide range of both common and edge cases to ensure the program's reliability and results. All of our tests can be found under the `tests/` directory.

To simplify the tests' execution, a simple bash script (`run-tests.sh`) is provided, with which all the unit tests are executed at once. This script is also used as an automated testing tool in our GitHub repository, whenever a push or a pull-request occurs (GitHub Actions).

Vectors class

The `Vectors` template class handles vector collections and provides quick and easy access to their distances.
Members:

- `vectors`: Represent both the base and queries vectors in a 2D array instead of `std::vector`, because profiling the program revealed that the operator[ ] of the `std::vector` significantly increased the execution time.
- `dist_matrix`: In this 2D array, the euclidean distances between all the vectors are cached/saved, for easy and quick access, since the calculation of the euclidean distance is a highly time-consuming operation.

Constructors:

- `Vectors(const std::string& file_name, int& num_read_vectors, int max_vectors, int queries_num)`: Loads vectors from a binary file, initializing `dist_matrix`.
- `Vectors(int num_vectors, int queries_num)`: Initializes `num_vectors` with generated values and caching euclidean distances. Used only for testing.

Methods:

- `size`: Returns the number of loaded vectors.
- `dimension`: Returns the vectors' dimension.
- `euclidean_distance`: Computes and caches the Euclidean distance between two vectors.
- `euclidean_distance_cached`: Retrieves the cached Euclidean distance.
- `query_solutions`: Retrieves k-nearest neighbor indices for a query from a file.
- `read_queries`: Loads multiple query vectors from a file, updating the distance cache.
- `add_query`: Adds a new query vector to vectors and updates distances. Used only for testing.

Directed Graph class

As already highlighted, the `Vectors class` provides an API that uses only indexes (integers) to refer to vectors. Leveraging this property, the `Directed Graph class` solely stores indexes and uses the provided API for all vectors-related functionality. The directed graph is implemented as an **array of unordered sets**. Each row of the array represents the corresponding vector-index, and the matching set contains the indexes of the vector's neighbors. Therefore, the neighbors of the vector with index 4 are in the unordered set: `neighbors[4]`.

This design choice maximizes throughput by offering $O(1)$ time complexity in the average case for all graph-methods, since unordered sets are implemented using **hash tables** as the underlying data structure. This implementation is also very straightforward, since all the methods just use the *unordered-set C++ API* to conduct the necessary operations. We firmly believe that each method's code is easy to grasp from the comments alone.

GreedySearch(G, s, x_q, k, L)

The code of greedy search closely follows the algorithm with the following modifications for performance optimization:

- `L_set`: Is represented by an ordered set according to euclidean distance from query, for easier access to the closest vectors.
- `visited`: Is represented by a boolean array for faster determination of index visitability.
- `return values`: As the visited set, the `L_set` is returned, which after the while-loop it contains only visited nodes. Due to the constraint on `L_set` being limited to a length of `L`, some visited nodes may have been deleted. Thus, before returning, using the `visited` array they are reinserted to `L_set`.

robust_prune(G, p, V, a, R)

Modifies graph 'G' by setting at most 'R' new out-neighbors for point with index 'p' and 'a' distance threshold exactly as shown in the pseudo code in Algorithm 2. The only difference is that 'V' is now an ordered candidate set containing pairs of (euclidean distance, index) and it is sorted in ascending euclidean distance between each point and point 'p'. We do this to calculate p^* in $O(1)$ time (it is the first element of 'V') instead of manually calculating it in $O(n)$ time. Moreover, all euclidean distances are cached inside our distance matrix, so the comparison $\alpha \cdot d(p^*, p') \leq d(p, p')$ is also done in $O(1)$ time.

Vamana Indexing Algorithm

The `vamana.hpp` file contains the implementation of the Vamana Indexing Algorithm, as described in the assignment's paper, as well as complementary functions. Specifically:

- `random_graph`: Generates a random R-regular directed graph. The R-regular property only concerns the out-degree of the vertices, as highlighted in the paper.
- `medoid`: Calculates and returns the medoid point of given dataset. A simple brute-force algorithm is used, as per instructions. The only slight optimization regards the use of `euclideian_distance_cached()` for the already calculated distances.
- `vamana`: Constructs a vamana graph using the aforementioned functions. This implementation is entirely based on the provided pseudocode.
- `write_vamana_to_file`: Stores a vamana graph into a (binary) file, for future use.
- `read_vamana_from_file`: Loads a vamana graph from a (binary) file. This function can be used as an alternate constructor to the `vamana()` function.

utils.hpp

Contains a useful utility function commonly found in most files: `ERROR_EXIT(cond, msg)` is a preprocessor macro which terminates the program and prints the given error message whenever the condition is true.

main.cpp

In the main function there are two sections of menus.
The first menu
— **Vamana Initialization** —

- 1) Read Vamana structure from file
- 2) Create a new Vamana structure

Enter choice (1-2):
which represents the user's choice to create the Vamana graph or read it from a file.

The second menu
— **Menu** —

- 1) Find k-nearest neighbors for a query
- 2) Find k-nearest neighbors for all queries
- 3) Save Vamana structure to file
- 4) Exit

Enter choice (1-4):
which represents the user's choice to find the k-nearest neighbors for one query or all. Printing the recall.