ESTAData - Algorithms and Data Structures

Nicolas Loza

1 Introduction

This document serves as documentation of the java implementation of the spatio-temporal clustering framework proposed in [1], under the scope of the ESTAData project, directed by Julio Borges. Therefore, it is recommended for the readers to be familiar with the terms and definitions presented in [1]. Moreover, the framework uses the Terracotta library for data storage and management. Thus, the reader should also bring basic knowledge of how said library works. Finally, the code corresponding to this documentation can be found in the repository `https://github.com/aqnmd/ESTAData`.

2 Overview

This section provides a quick overview of the project's packages and the functionalities they provide.

- `Default`: only the entry point class `Mining.java` is provided, which parses the input provided through the console and interprets it accordingly.
- `de.estadata.mining.datatransformation`: contains three important classes:
  - `BigMemory.java`: instantiates the Terracotta CacheManager.
  - `Report.java`: represents a report, thus containing properties like coordinates, timestamps and description, among others.
  - `Cluster.java`: represents a set of Reports that have been clustered together (how is not specified).
- `de.estadata.mining.gephiextension`: contains classes that fix minor bugs found in the Gephi library.
- `de.estadata.mining.graphmodel`: models the graph structure used by all of the framework's algorithms. Consists of:
  - `Node.java`: models a single node in the graph (that corresponds to a single report).
  - `Edge.java`: models an edge between two nodes (meaning the respective reports are ST-connected).

  - `Graph.java`: manages the sets of nodes and edges by storing them in separate Terracotta Caches.
  - `GraphView.java`: models a 'view' of the graph by containing a subset of the graph's nodes and/or a subset of its edges.

- `de.estadata.mining.modularityoptimizer`: provides a set of modularity based clustering algorithms. It is an adaptation of the code provided in `http://www.ludowaltman.nl/slm/`.
- `de.estadata.mining.scan`: provides a single class `SCAN.java` implementing the algorithm of the same name on the previously mentioned graph structure.
- `de.estadata.mining.util`: provides utility functionalities divided in two classes:
  - `DataLoader.java`: is the class that allows the loading of reports data from .csv and .json files.
  - `MiningTools.java`: is a class providing a wide range of methods that are used throughout the framework, but can also be useful for external users of the library.

The project is also provided in form of a jar file, which can be used directly to execute most of the functionalities made available by the packages mentioned above. We will refer to this file from now on as "`mining.jar`", and in the following sections will provide examples of how to used under different circumstances and for different purposes.

3 Data Loading

Once Terracotta has been correctly set up (see `terracotta.org`), we can proceed to the first step: loading the data into Terracotta Caches. Using `mining.jar` this is quite straightforward:

```
$ java [JVM args] -jar mining.jar --config CONFIG_FILE --reportscache
CACHE --load PATH --type [json|csv] --ratio R
```

As we will see in future sections, the JVM arguments tend to have a big impact in the performance of the framework. Especially, when loading big datasets, the JVM should have access to as much memory as possible. This is done using the flag `-Xmx` (e.g. `-Xmx8G` will allow the JVM heap to use up to 8G of memory). Regarding the program arguments

- CONFIG_FILE is the path to the Terracotta configuration file,
- CACHE is the name of the Terracotta cache where the reports should be stored in,

- `PATH` is the path to a directory or single file containing the data (must be `json` or `csv`),
- and the ratio `R` is a parameter in range $[0, 1]$ signalizing which percentage of the data should be loaded. Its default value is 1.0, and if minor to one, the selected data is random (i.e. two executions using the same ratio could result in different sets of data being loaded).

It is worth noting that the flags `--config` and `--reportscache` are not specific to the load functionality, but must *always* be provided when using `mining.jar` from the command line.
In case of using the framework as a library, the same functionality is available under the aforementioned class `DataLoader`, which provides a single static method `load(String dataType, String dataPath, Cache reportsCache, double ratio)`. The names of the arguments represent the same as the above command line arguments.
In order to successfully load the information from the csv/json files, they need to provide certain information. Otherwise, the load functionality ends in failure. This is avoided if the data files contain the following columns with their respective values:

- 'lng' for longitude (a double precision floating point number).
- 'lat' for latitude (a double precision floating point number).
- 'created_at' for the creation time of the report. For .csv files, the date formatting should be 'yyyy-MM-dd HH:mm:ss', for .json files: 'yyyy-MM-dd'T'HH:mm:ss' (a string).
- 'summary' for the category of the report (a string).
- 'description' for the user's description of the report (a string).
- for the report's URL address, 'bitly' in case of .csv files, 'html_url' in case of .json files (a string).
- 'id' for the report's ID (an integer).
- in case of .json files, 'reporter' for the ID of the reporter (an integer).

4 Graph generation

Because of the nature of the different analysis provided in the framework, we first need to generate a graph from the set of reports previously loaded into the cache. Fortunately, this process is mostly automatized and can be called in a single command line:

```
$ java [JVM args] -jar mining.jar [config+cache] --mode filter -m M
-d D
```

As when loading data into Terracotta, it is important to provide the JVM

with as much heap memory (flag `-Xmx`) as well as with off-heap memory (flag `-XX:MaxDirectMemorySize`). Regarding the program arguments:

- *`config+cache`*: here, the configuration file of terracotta and the cache containing the reports needs to be provided (see 3). However, a further cache is also needed. In it, the clustering results are to be stored. This cache is provided using the flag `--clusterscache`.
- `--mode filter` indicates the modality being executed. In future sections we shall see the alternatives.
- `-m M` or `--meters M` indicates the maximal spatial distance M in meters that two reports can have to be ST-connected.
- `-d D` or `--days D` indicates the maximal temporal distance D in days that two reports can have to be ST-connected.

The successful execution of this command line has the following results:

- Two new caches generated containing the information concerning the graph structure. These are named `[reportsCache]_nodesCache` and `[reportsCache]_edgesCache`. Thus, if the cache containing the reports is called `NYC`, the new caches are called `NYC_nodesCache` and `NYC_edgesCache`. As we can see from their names, the first cache contains the graph's nodes and the second one its edges.
- Each node in the graph represents a single report, and an edge between two nodes means they are ST-connected under the provided parameters.
- Once the graph has been generated, each of its connected components is assigned a unique, positive cluster ID (isolated nodes are assigned the cluster ID -1).
- Once every node in the graph has a cluster ID, they are grouped in `Cluster` objects according to their cluster ID. These `Cluster` instances are then stored in the corresponding Terracotta cache.

When using the framework as library, the same functionality is provided by the class `STFiltering`. Unlike the command line version of the framework, the usage of this class to achieve the same results as the ones listed above requires several steps. We shall now provide an overview of the class' constructor and methods, and then some examples as how to use them.
When creating an instance of `STFiltering`, there are three possible ways to do so:

1. `STFiltering(List<Integer> reportIDs, CacheManager databaseManager, Cache reportsCache, int maxSpaceDist, int maxDayDist)`
2. `STFiltering(CacheManager databaseManager, Cache reportsCache, int maxSpaceDist, int maxDayDist)`

3. `STFiltering(CacheManager databaseManager, Cache reportsCache, boolean useExistingGraphStructure)`

All of these constructors result in the generation of the graph's `Node` instances, but *not* the `Edge` instances (this takes place in a later step)[1]. As with the command line version, a Terracota `CacheManager` and a `Cache` need to be provided for the graph generation. Each constructor, however, offers a slightly different functionality. The first one allows its caller to specify a list of the report IDs to be used for the graph generation (i.e. every report whose ID is not contained in the list is ignored). The second one does not take any list of IDs and works with all the reports present in the provided cache. The third constructor has different behaviours depending on its third argument: if it is set to `true`, the program attempts to recover a previously generated graph instance corresponding to the given `Cache`. In this case, the respective caches for nodes and edges should exist under the given `CacheManager` and be named as previously specified. On the other hand, if the third parameter is set to `false`, the program reads all reports from the given cache and generates a graph using standard values for spatial and temporal distance: 100 meters and 30 days, respectively.
There is a very particular side-effect from running any of this constructors, except for the third one for recovering a previously generated graph. Namely: the program generates a set of temporary Terracotta caches (hereafter referred to as "buckets"). How many, depends on the total time span of the reports and on the maximal temporal distance two reports may have. More precisely[2]:

$$timeSpan = endTime - startTime$$

$$\#buckets = \left\lceil \frac{timeSpan}{maxDayDist} \right\rceil$$

These buckets store information about the reports and help in the generation of the graph's edges, which is the next step after constructing the instance. This is achieved by calling the method `void generateGraph()`. More importantly, because of the algorithms quadratic complexity, the buckets help avoiding the worst case scenario of comparing each node to every other node in search for possible ST-connections. Instead, only nodes that are in close temporal proximity are compared with one another. Hence the creation of the buckets. Moreover, because the set of report references contained in each bucket is disjoint with every other bucket, this allows a very effective parallelization of the algorithm. For specific results, see **??**. Finally, once the graph's edges have been generated (i.e. once the execution of `generateGraph()` is finished), all buckets are disposed of.
After generating the graph, one can directly proceed to generate the clusters from the graph's connected components and transfer them to a separate `Cache`.

[1] Despite the edges not being yet generated, the corresponding `Cache` instances for nodes and edges are generated after the execution of each constructor.

[2] Because *timeSpan* is given in milliseconds, *maxDayDist* must also be translated to milliseconds.

This is done by calling the method with the signature `void generateAndTransferClusters(Cache clustersCache)`. Note that this method clears any previous data stored in the `Cache` passed as argument and writes its results. However, the class also offers an alternative method called `void generateAndTransferClusters(Cache clustersCache, GraphView view)`, which does has the same behaviour as the previous one, except that it only clusters nodes and edges whose IDs are contained in the passed `GraphView`.
Such a view can be generated using the method `GraphView filter(double newMaxSpaceDist, int newMaxDayDist, boolean mustShareCategory)`, which allows its caller to execute a filtering over the graph's edges with more strict parameters as the ones used for generating it. The resulting view contains all of the graph's node IDs, but only the edge IDs of those edges that fulfil the more strict parameters.

5 Graph clustering

Once the graph has been generated (see 4), the framework also allows the usage of known clustering algorithms for the refinement of the clusters given by the graph in a first instance (which are simply its connected components). From the command line, these algorithms can be executed as follows:

```
$ java [JVM args] -jar mining.jar [config+caches] --mode cluster --algorithm
ALG [ARGS]
```

Like in 4, the JVM arguments should assign the as much memory (on- and off-heap) as possible to the program. Just as well as before, *config+caches* contains the necessary information about the Terracotta configuration file and the caches containing the reports and clusters. As in some cases of the previous section, in this scenario we need a graph to work on. Therefore, the necessary caches should exist with the convention names (see 4 for further information).
Regarding the remaining arguments, `--mode cluster` indicates that the program must execute a clustering algorithm over the (presumably existent) graph; `ALG` is the algorithm to be executed, which can be one of the following: `scan` (for the algorithm of the same name, see [2]), `louvain`, `louvain_mlv` (for Louvain with multilevel refinement) or `slm` (for smart local moving algorithm for large-scale modularity-based community detection). The last three are modularity based algorithms, and their provided implementations were taken from `http://www.ludowaltman.nl/slm/`.
When using SCAN, `ARGS` consists of: `--mu MU --eps EPS`, which are the algorithm's parameters as described in [2]. If not provided, `MU` and `EPS` take the default values 2 and 0.7, respectively.
On the other hand, when using any of the modularity based algorithms, a wide set of arguments is available for the user.

- `--modularity_function FUNC`: the modularity function to be used (`standard` (default) or `alternative`).

- `--resoltion RESOL`: the resolution parameter (default: 1.0).
- `--random_starts RANDS`: the number of random starts executed by the algorithm, default: 10.
- `--iterations ITER`: the number of iterations per random start, default: 10.
- `--random_seed SEED`: seed for the RNG, default: random.

For more detailed information about these arguments, we refer to the aforementioned website.

When using the framework as library, the same functionalities are made available by the class `GraphClustering`. In order to use it, one needs to instantiate it by calling `GraphClustering(graph, pointsCache, clustersCache)`. The specific algorithms are run either by calling `void runSCAN(epsilon, mu)` to run SCAN, or by calling `void runModularityOptimizer(modFunc, resolution, algorithm, randomStarts, iterations, randomSeed)` for any of the other three algorithms. Finally, once the chosen algorithm has finished its execution, `generateAndTransferClusters()` allows the generation of `Cluster` instances and their storage in the respective `Cache`.

References

[1] Budde, M., Borges, J. D. M., Tomov, S., Riedel, T., & Beigl, M. Improving Participatory Urban Infrastructure Monitoring through Spatio-Temporal Analytics.

[2] Xu, Xiaowei, et al. "Scan: a structural clustering algorithm for networks". Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2007.