Incremental DBSCAN for Green Computing

Jorge Reis and Mário Ulisses Costa
VisionSpace Technologies
Rua Alfredo Cunha, 37
4450–023 Matosinhos, Portugal
{*jorge.reis, ucosta*}*@visionspace.com*

***Abstract*—The Green Computing area is a global trend that concerns the impact of the computational resources in the environment. These field is becoming more relevant today since the exponential growth of the Information Technology sector, demanding increasingly more energy and thus polluting in the same proportion. In previous studies, scientists concluded that nearly 90% of the energy consumption in a computer system is due to software execution [1].**

This paper refers to the work developed under the project GREENSSCM: Green Software for Space Control Mission[1], which aims to reduce the energy consumption in software execution by predicting the best transformations and optimizations to apply in the source code. Using a developed static analyzer and a set of code repositories we extract software metrics from each atomic piece of code (for instance a function or a method) and use the clustering technique to automatically group them all by their degree of similarity. Later, we monitor the energy consumption of the execution for the same code with a set of different optimizations. Then we relate these with the metrics extracted previously and discover what are the best optimizations for the atomic piece of code of each cluster.

This paper also presents and describes the implementation of an incremental DBSCAN algorithm which allows us to add new code repositories to a previous clustered database and its integration into the ELKI (Environment for Developing KDD-Applications Supported by Index-Structures)[2] Framework.

***Keywords*-Green Computing; Artificial Intelligence; Clustering; Incremental DBSCAN**

I. Introduction

The concern with the energetic consumption of computing resources is growing. Much has already been done in order to reduce the consumption in hardware devices, for example, using virtualisation and replace old hard drives to Solid State Drives in physical computers.

This effort has been so successful that it was estimated that 90% of energy consumption with respect to an ICT system is due to the software. So we face a different challenge now: optimize the energy consumption of a software system.

In the context of GreenSSCM project we have designed and implemented a statical and dynamic source code analyser that together can discover the most suitable compiler optimisations a piece of software (For the purpose of this paper we consider a piece of software as a function/method). Our statical analyser is designed to extract metrics from C++ source code, convert them into characteristics for the DBSCAN algorithm, with the distance function and store the results from the classification. The dynamic analyser uses the RAPL (Running Average Power Limit) Intel framework[3] to extract the measurements of energy consumption of a particular piece of software and compare the execution of the software with and without compiler optimizations.

We use the results from the statical analyser to cluster pieces of software according to their similarity and the dynamic analysis to correlate the compiler optimizations to the clusters. From the beginning we decided to use an Off-the-shelf framework that implements clustering algorithms. We have experimented several frameworks, but the one that was more suitable was ELKI because of their degree of configuration (both using the ELKI user interface and the API). We have experimented several clustering algorithms, such as: K-Means, OPTICS and SUBCLU, but DBSCAN seemed to us to be the most interesting since:

- It does not require one to specify the number of clusters in the data *a priori*;
- Can find arbitrarily shaped clusters;
- Has a notion of noise;
- Requires just two parameters and is mostly insensitive to the ordering of the points in the database;
- Is designed for use with databases that can accelerate region queries.

Since we are interested in classify several software packages (each one with several thousands of functions) we need an incremental DBSCAN implementation in the ELKI framework. We found out that no incremental DBSCAN implementation was available much less in ELKI.

In this paper we explain in more detail our requirements regarding the GreenSSCM project, how we implemented the Incremental DBSCAN algorithm and the integration with ELKI framework.

This work is partly funded by the Innovation Agency, SA, Northern Regional Operational Programme, Financial Incentive Grant Agreement under the Incentive Research and Development System, Project No. 38973.

[1]More information at: http://green.visionspace.com

[2]More information at: http://elki.dbs.ifi.lmu.de/

[3]More information at: http://tinyurl.com/o674ro2

II. Software Metrics

The first part of the work was focused on search and define a set of software metrics that was relevant to characterize and identify source code, in this case, for functions/methods. These metrics should be as stable as possible in a way that similar functions have similar metric values. With this principle in mind, we have gathered a substantial set of metrics, analyzing its connection with energy consumption. So, we started by considering the number of parameters and returns, but as these characteristics may not be directly associated with energy consumption and we can have similar functions with a different number of parameters and returns, we discarded these. The interface complexity, the sum of the number of parameters and returns, was also logically ignored. In terms of number of lines of code we can extract LOC (total lines of code), ELOC (effective lines of code) and LLOC (logical lines of code). As LOC and ELOC are directly dependent of the programmer's writing, we cannot relate it with energy consumption. The ELOC metric, as it represents only the number of executable statements independently of the code format, are considered then. One metric we can directly associate with time execution and energy consumption is the Cyclomatic complexity (McCabe's Complexity) because it indicates the number of linearly independent paths within a piece of code. Slightly related with the cyclomatic complexity, we extracted the number of each different type of statements: conditional *if/else*, inlined *if-else*, *for* and *while* loops, logical *and* and *or*, and switch case). Another complexity measurement is the functional complexity, the sum of the cyclomatic and interface complexity. As we have discarded interface complexity, we also did not consider functional complexity.

Finally, we have extracted the shape of the source code. The shape is a kind of structure, representing the main statements and its hierarchy in a piece of code. For instance, looking at Figure 1, we can see how the source code 1a is transformed into the shape at 1b.

So, despite the existence of many software metrics, only some have been chosen. Just the following were considered:

- Number of Logical Lines of code - LLOC (number of executable "statements" (that end in semicolon));
- Cyclomatic complexity (McCabe's complexity);
- Number of conditional $if/else$ statements;
- Number of inlined $if - else$ statements;
- Number of for statements;
- Number of $while$ statements;
- Number of logical and statements;
- Number of logical or statements;
- Number of *switch case* statements;
- *Shape* (structure) of the code.

III. Clustering Application

Following the main idea behind this project, the final goal is to have an application that receives one or more source code repositories, extracts the metrics described above for each function, and then groups the functions by their degree of similarity.

```
int i = left, j = right;
int tmp;
int pivot = arr[(left + right) / 2];

while (i <= j) {
    while (arr[i] < pivot)
          i++;
    while (arr[j] > pivot)
          j--;
    if (i <= j) {
          tmp = arr[i];
          arr[i] = arr[j];
          arr[j] = tmp;
          i++;
          j--;
    }
};

if (left < j)
    qs(arr, left, j);
if (i < right)
    qs(arr, i, right);
```

(a) Source code

```
= ; = ;

=(+ ; /) ;

while(
    while(
       ++) ;
    while(
       --) ;
    if(
       = ;
       = ;
       = ;
       ++ ;
       --
    )
) ;

if(
    call ) ;
if(
    call ) ;
```

(b) Shape

Figure 1: Example of a shape representing a piece of code.

Although we use large C++ repositories in order to improve the clusters detection and the knowledge base, our main objective is to improve the energy efficiency of the SCOS-2000 (Satellite Control and Operation System)[4]. SCOS-2000 is a satellite mission control system developed and maintained by the European Space Agency (ESA/ESOC) and provides the means for operators to monitor and control satellites. SCOS is written in C++ language, has more than a million lines of source code and over 31 thousand functions/methods.

Having a database populated with several functions and software metrics from a repository, we group these functions by their similarity. Looking for a way of identify these groups in the most "intelligent" and automatic way (analysing the existing artificial intelligence branches), Clustering technique seems to be the most appropriate one.

There are multiple machine learning software tools available. Weka is one of them, but the use of its DBSCAN implementation is discouraged by the development team. Java-ML is another Java-based option, but it uses Weka algorithms for clustering. Apache Mahout is another powerful tool, but it only offers K-Means for clustering. The most versatile and complete tool we found was ELKI Data Mining Framework [2], developed in Ludwig-Maximilians Munich University and therefore the one used in that project. It is very configurable and has visualisation tools included.

As is well known, clustering has several implementations.

[4]More information at: http://www.esa.int/Our_Activities/Operations/gse/SCOS-2000

Although there are different models, these algorithms are basically differentiated by their input.

We have tried the K-Means algorithm, but although it is very efficient, it must receive a number of desired clusters. The same happens with the EM algorithm. As we have no idea in our data model of the number of clusters needed to group the functions, and knowing that it tends to vary according to the functions we are using. We also tried SUBCLU, a subspace clustering algorithm for high-dimensional data. Based on DBSCAN algorithm, SUBCLU can find clusters in subspaces, using a bottom-up strategy. However, we cannot parametrize the number of sub desired dimensions and it leads to a nonsensical amount of clusters because it combines all the permutations of the dimensions (our software characteristics). Testing the algorithm with 40.000 functions, we found it non computable after running for 48 hours in a 10GB RAM machine.

A. DBSCAN

After some experiments, we found in DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm the best solution for our problem. This algorithm, first introduced by Ester, et al. [3], identifies clusters of elements in a given data set by looking at the density of points, with the advantage of also identify noise, i.e. elements not associated with any cluster.

This algorithm receives two values as parameters: ϵ (epsilon, the radius that delimitates the neighbourhood area of a point) and *minPts* (minimum number of points required to form a dense region). The global idea of the algorithm is that the neighborhood, for a given radius (ϵ), of each point in a cluster has to contain at least a minimum number of points (minPts). The points in the database are then classified as *core points*, *border points* and *noise points*, related with the density relations between points to form clusters. A core point p is a point that belongs to the global data set D and has at least *minPts* points in its ϵ-neighborhood, formally defined as: $p\dot{\in}D$. Then, if p is a core point, it forms a cluster with the points that are reachable from it.

$N_\epsilon(p)$ is the subset of D contained in the ϵ-neighborhood of p. The points contained in $N_\epsilon(p)$ are said to be directly density reachable from the core point p, formally defined as:

$$p \overset{D}{\twoheadleftarrow} q \overset{def}{=} p \in N_\epsilon(q) \wedge Card(N_\epsilon(q)) \geq MinPts$$

So, an object p is directly density reachable from an object q if p is in the ϵ-neighborhood of q and q is a core point (the number of points in its ϵ-neighborhood is greater than *MinPts*).

Border points are non-core points but density-reachable from another core point, while noise points are also non-core points and not density-reachable from other points, do not belonging to any cluster. A point q is density reachable from a point p if there is a path between p and q, where all the points on the path (with possible exception of q) are core points, i.e. in a path $p_1, ..., p_n$ with p_1 representing p and p_n representing q, each p_{i+1} is directly reachable from p_i. The formal definition is presented below:

$$p >_D q \overset{def}{=} \{p_1, \ldots, p_n | p_1 = q \wedge p_n = p \wedge p_i \in D \wedge p_{i+1} \overset{D}{\twoheadleftarrow} p_i\}$$

As default, DBSCAN uses the euclidean distance function, commonly used to calculate the distance between points in a N-dimensional space. However, as this function is only appropriated for numerical values, we need a solution that includes a calculation of the distance between the shapes, represented by a *string* value.

Taking advantage of the tree structure of the shape, we used the tree edit distance, similar to Levenshtein distance for strings, to measure the difference between two trees. Informally, this distance is defined as the minimum-cost sequence of node edit operations (i.e. insertions, deletions and substitutions) that transform one tree into another. So, the Robust Tree Edit Distance [4] implementation is used to calculate the distance between the functions' shapes and the euclidean distance used to calculate the global distance between two points, taking into account all the metrics assumed. An example of a distance result between two shape trees, following this approach, can be seen in figure 2. For that example case, we have a value 2 of distance due to a substitution ($for \rightarrow cal$) and a deletion ($call$ from first shape).

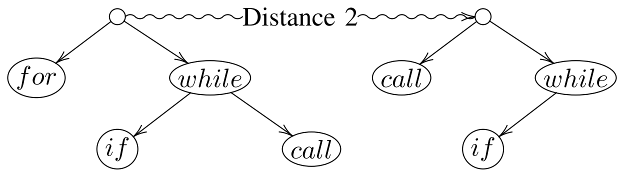

Figure 2: Example of a distance between two shapes using RTED

Looking at the euclidean distance function and the metrics used, we found that all the metrics have the same weight in the distance calculation. But, reflecting about it, we think that we can assume that some metrics may influence more or less the distance between functions in terms of energy consumption. So, we found useful to consider that each metric can have an independent weight in the distance formula. Thus, we adapt the euclidean equation 1 to consider, for each dimension, a weight value greater than zero. The new distance function used in the DBSCAN algorithm is represented in the equation 2. With this change, we can keep some less relevant software metrics without compromise the clusters detection.

$$d(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + ... + (q_n - p_n)^2} \quad (1)$$

$$wd(p,q) = \sqrt{(q_1 - p_1)^2 * w_1^2 + (q_2 - p_2)^2 * w_2^2 + \dots \dots + (q_n - p_n)^2 * w_n^2} \quad (2)$$

IV. Incremental DBSCAN

DBSCAN, as talked before, is applied to a static database. Although it meets the objectives, for our project is highly desirable to perform the clustering update incrementally. When we want to add some new repository or just classify a new piece of code, we had to rerun the algorithm for all the accumulated database. Due to the density-based nature of DBSCAN, the insertion of a new object affects the current clustering only in the neighborhood of this object. Then, the implementation of an efficient incremental DBSCAN algorithm for insertion of new functions to the existing clustering database is described in this sections of this paper.

The implementation of Incremental DBSCAN was based in the definitions provided in [5] which proves that the incremental algorithm yields the same result as the non-incremental DBSCAN algorithm, based on the formal notion of clusters. So, we can determine which part of an existing clustering process is affected by an insertion of a new point. And for that we focus on two main definitions: *affected objects* and *update seeds*.

A. Affected Objects

On an insertion of a new object p in the database D, if there exists new density connections, non-core objects in $N_\epsilon(p)$ may become core objects. So, the set of *affected objects* represents the objects which may potentially change cluster membership after the insertion, while the cluster membership of other objects not in this set will not change. Thus, we only have to consider these objects in order to process the update. On the new insertion, the set of *affected objects* is the set of objects in $N_\epsilon(p)$ plus the objects density reachable from one of these objects, formally defined as:

$$Affected_D(p) \overset{def}{=} N_\epsilon(p) \cup \{q | \exists o \in N_\epsilon(p) \land q >_{D \cup \{p\}} o\}$$

B. Update Seeds

However, it is not necessary to reapply DBSCAN for all the *affected objects* set. We can process only over certain *seed* objects, ϵ-neighbors of the point to be inserted that are core objects after the insertion. As it is not necessary to rediscover density connections known from the previous clustering, we finally only have to look at core objects in the ϵ-neighborhood of the objects in $N_\epsilon(p)$ that change their core object property. To get these objects more efficiently, when we run an initial DBSCAN we also store in the database the number of ϵ-neighbors of each point. Thus, we only need to call a region query for the new inserted object to determine all the objects q' with a changed core object property. These are the objects with number of neighbors equals to $MinPts - 1$. Lastly, for these objects q' we have to retrieve $N_\epsilon(q')$ to get all objects in the $UpdSeed_D(p)$.

The set $UpdSeed$ is formally defined as:

$$UpdSeed_D(p) \overset{def}{=} \{q | q \dot{\in} D \cup \{p\}, \exists q' : q' \dot{\in} D \cup \{p\} \land \land q' \dot{\notin} D \land q \in N_\epsilon(q')\}$$

C. Insertion Results

When we insert a new point p into the clustering database D, and after determine the set $UpdSeed_D(p)$, we can deal with one of the following cases:

1) Noise
   The calculated $UpdSeed_D(p)$ is empty, so there are no *new* core points. As a result, just p is assigned as a noise point.
2) Creation
   In this case, $UpdSeed_D(p)$ contains only core objects not belonging to any cluster in the previous clustering, i.e. classified as noise. As a result, a new cluster is created containing p and these seed objects.
3) Absorption
   $UpdSeed_D(p)$ contains core objects belonging to exactly one cluster C, in the previous clustering state. The new point p and possibly noise points in the $UpdSeed_D(p)$, if exists, are absorbed into cluster C.
4) Merge
   If $UpdSeed_D(p)$ contains core objects belonging to more than one cluster before the update. In this case, all the clusters are merged into one new cluster as well as the new point p.

So, for each function inserted we follow the previous steps until have an insertion result. At this point, upon the result case, the clustering database is updated the most efficient way. Thus, only the cluster assignment of the new point p and the changed objects were updated in the database. To maintain the number of ϵ-neighbors of each element up to date in the database, we also increment by one the number of neighbors of the elements in the set $N_\epsilon(p)$.

D. Performance Evaluation

Considering the objective of insert and classify a single function when we already have a clustered database, the Incremental DBSCAN brings us a great advantage. Having a database with 20500 functions already clustered, if we want to add and classify a new function, it takes more than 66 minutes to rerun DBSCAN. However, using the Incremental DBSCAN algorithm, even considering that the previous functions are already clustered, it takes only 43 seconds, in average, to insert a new point in a database of that size. In this analysis we realize that the execution time for an insertion tends to increase with the increase of the database because there was more *affected objects* to process. But the density of the region where the point will be located determines how many operations are necessary and consequently the run time needed. Even when we add a new repository to our database and we want to cluster those new functions, if its size represents a low percentage of the already clustered database, it may be faster to perform an Incremental DBSCAN for those points.

In the figure 3 we have a chart that represents, for an already clustered database of 20.500 functions, the execution time cost to add a new portion of functions (could be only one). As the DBSCAN algorithm does not allow increments, we have to rerun the entire clustering process with all the desired functions: the old ones already clustered and the new ones. So, it implies an execution time superior to the previous clustering. However, running the developed incremental DBSCAN algorithm, we have a much faster process when adding a reduced amount of new functions. So, the green line in the chart represents the accumulated time to add several functions, one by one. Even though, for this specific tested dimension, we can figure out that up to an insertion of 80-100 new functions, the Incremental DBSCAN algorithm is more efficient to rerun DBSCAN.

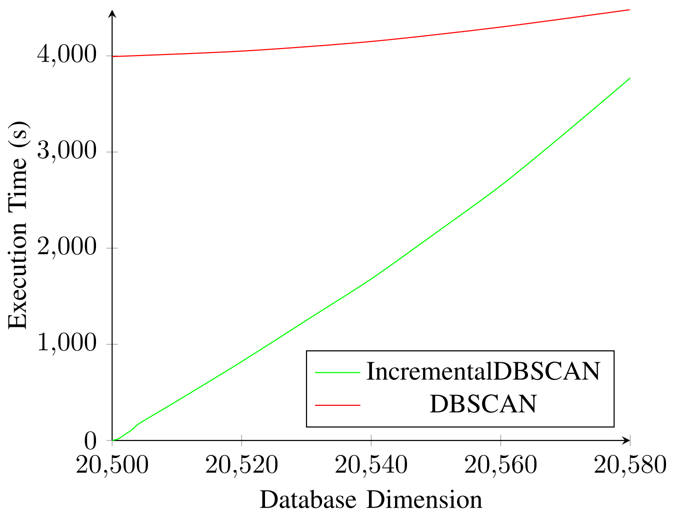

Figure 3: DBSCAN vs IncrementalDBSCAN chart

The tests and the execution times shown were obtained using a machine with the specifications described in the table I.

Component	Detail
Operating System	Ubuntu 14.04 LTS
Processor	Intel® Xeon(R) CPU E5645 @ 2.40GHz x 8
RAM	9.8 GB

Table I: Computing platform configurations

V. ELKI Integration

The Incremental DBSCAN algorithm described above was implemented according to the ELKI's available API and integrated in the platform. We make use of all the facilities and explore the modular structure of this framework. To deal with stores and updates we develop a database connection that allows a direct treatment of the data. In respect to the algorithm, we make use of the ELKI's internal static database and use the $R^\star$-tree queries to obtain the objects in the ϵ-neighborhood of a given point. The results are processed according to one of the four explained possible results, directly updating our external database.

VI. Future Work and Conclusions

We will contribute with this implementation for the ELKI framework, so it can be used by anyone. ELKI has been shown to be an extremely well organized tool, very modular and extensible as it permits the integration of custom components, adapted to our needs. Following the ELKI structure, and using its API, it's only necessary to create a module with the core of the algorithm and put in the existence structure. The parameterization is also simple, as the parameters are passed using the existing command line interface and automatically handled and checked by the platform. To deal with the results we have developed several result handlers too, one for each case, and fit them into the respective ELKI component. As clustering in ELKI was developed for a complete and independent process, in incremental DBSCAN each new insertion has to be an independent process and consequently the ELKI internal database has to be initialized for every run. When incrementally clustering more than just one new function, it would be very useful, as an improvement, to do a incremental clustering in just one ELKI call, avoiding the repetitive initialization of the database and expectedly a performance boost.

In terms of the *GreenSSCM* project, as mentioned at the beginning of this paper, the goal is to correlate the energy optimizations at the dynamic analysis with the clusters obtained in the static analysis. Later, when inserting a new point to the clustering database with the incremental DBSCAN we get a cluster with some compiler optimizations associated. Like so, we are able to categorize future functions just by doing a table lookup on the database of the already clustered data set.

References

[1] G. G. Protocol, *GHG Protocol Product Life Cycle Accounting and Reporting Standard ICT Sector Guidance*. GreenHouse Gas Protocol, 2013.

[3] M. Ester, H. peter Kriegel, J. S, and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise.” AAAI Press, 1996, pp. 226–231.

[4] M. Pawlik and N. Augsten, “Rted: A robust algorithm for the tree edit distance,” *Proc. VLDB Endow.*, vol. 5, no. 4, pp. 334–345, Dec. 2011. [Online]. Available: http://dx.doi.org/10.14778/2095686.2095692

[5] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu, “Incremental clustering for mining in a data warehousing environment,” in *Proceedings of the 24rd International Conference on Very Large Data Bases*, ser. VLDB ’98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 323–333. [Online]. Available: http://dl.acm.org/citation.cfm?id=645924.671201

[2] E. Achtert, H. Kriegel, E. Schubert, and A. Zimek, “Interactive data mining with 3d-parallel-coordinate-trees,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, 2013, pp. 1009–1012. [Online]. Available: http://doi.acm.org/10.1145/2463676.2463696