Moving to Smaller Libraries via Clustering and Genetic Algorithms

G. Antoniol[*], M. Di Penta[*], M. Neteler[**]
antoniol@ieee.org, dipenta@unisannio.it, neteler@itc.it

[(*)] RCOST - Research Centre on Software Technology
University of Sannio, Department of Engineering
Palazzo ex Poste, Via Traiano, I-82100 Benevento, Italy

[(**)] Irst-ITC Istituto Trentino Cultura
Via Sommarive 18
38050 Povo (Trento), Italy

DRAFT

Abstract

There may be several reasons to reduce a software system to its bare bone removing the extra fat introduced during development or evolution. Porting the software system on embedded devices or palmtops are just two examples.

This paper presents an approach to re-factoring libraries with the aim of reducing the memory requirements of executables. The approach is organized in two steps. The first step defines an initial solution based on clustering methods, while the subsequent phase refines the initial solution via genetic algorithms.

In particular, a novel genetic algorithm approach, considering the initial clusters as the starting population, adopting a knowledge-based mutation function and a multi-objective fitness function, is proposed.

The approach has been applied to several medium and large-size open source software systems such as GRASS, KDE-QT, Samba and MySQL, allowing to effectively produce smaller, loosely coupled libraries, and to reduce the memory requirement for each application.

Keywords: library re-factoring, clustering, genetic algorithms

1. Introduction

Software miniaturization deals with a particular form of re-factoring aiming to reduce some measures of the size of a software system. Consider, for example, embedded systems; the amount of resources available is often limited, and thus developers are interested to reduce the footprint of executables. Applications running on hand-held devices have similar, even if less stringent, resource requirements. All in all, it is not infrequent that, as described in [7], the software *extra fat* needs to be eliminated or reduced. Clearly, several actions may be taken. First and foremost, dead code and software clones should be removed. Furthermore, some form of restructuring, at library and at object file level, may be required. The latest intervention must deal with dependencies among software artifacts.

For any given software system, dependencies among executables and object files may be represented via a dependency graph, a graph where nodes represent resources and edges the resource dependencies. Each library, in turn, may be thought of as a sub-graph in the overall object file dependency graph. Therefore, software miniaturization can be modeled as a graph partitioning problem. Any graph partitioning (into subgraphs) represents a problem solution characterized by the resource used by each executable present in the system. Unfortunately, it is well known that graph partitioning is an NP-complete problem [9] and thus often heuristics are adopted to find a sub-optimal solution. For example, one may be interested to first examine graph partitions minimizing cross edges between sub-graphs corresponding to libraries. More formally, a cost function describing the restructuring problem has to be defined, and heuristics driving the solution search process must be identified and applied.

In [7], a process to miniaturize software systems has been proposed. The central idea is to apply clustering techniques to identify software libraries minimizing the average executable size. Doval et al. [8], applied Genetic Algorithms (GA) to find what they called *meaningful partitions* in a graph representing dependencies among software components. Other communities, such as the optimization community, addressed the graph partitioning related problems in several ways. For example, constraints were incorporated by modifying the problem definition. To speed up the search process, heuristics based on GA and modified GA [27] were proposed.

This paper stems from the observation that previously proposed approaches to software miniaturization were not completely satisfactory. For example, it is not obvious if pruning clones may be beneficial to reduce the memory requirements of executables. Moreover, an approach based solely on clustering may be unable to find solutions easily identified by GA. Conversely, GA requires a starting population; choosing a random solution may not be very efficient, or it may lead to a local sub-optimal solution. To overcome the aforementioned limitations, we propose a novel approach where an initial sub-optimal solution to library identification (i.e., a set of graph partitions) is determined via clustering approaches, then followed by a GA search aimed at reducing the inter-library dependencies. GA is applied to a newly defined problem encoding, where genetic mutation may lead sometimes to generate clones, clones that do indeed reduce the overall amount of resources required by the executables, in that they remove inter-library dependencies.

Moreover, a multi-objective fitness function was defined, trying to keep low, at the same time, both the number of inter-library dependencies and the average number of objects linked by each application.

The approach was applied to improve the *GRASS* re-factoring presented in [7], and, to gain more empirical evidence, to other open source software systems such as *KDE-QT*, *Samba* and *MySQL*.

The paper is organized as follows. First, the essential background notions to help the reader are summarized in Section 2; the re-factoring approach and support tools are described in Section 3. Information on the case study systems is reported in Section 4. Section 5 presents case studies results. Finally, an analysis of related work is reported in Section 6, before conclusions and work-in-progress.

2. Background Notions

To re-factor the software systems libraries, clustering and GA were integrated in a semi-automatic, human-driven re-factoring process.

Clustering deals with the grouping of large amounts of things (*entities*) in groups (*clusters*) of closely related entities. Clustering is used in different areas, such as business analysis, economics, astronomy, information retrieval, image processing, pattern recognition, biology, and others.

GA come from an idea, born over 30 years ago, of applying the biological principle of evolution to artificial systems. GA are applied to different domains such as machine and robot learning, economics, operations research, ecology, studies of evolution, learning and social systems [10].

In the following sub-sections, for sake of completeness, some essential notions are summarized. Describing the different types of clustering algorithms or the details of GA is out of the scope of this paper. More details can be found in [2, 14, 16] for clustering and in [10] for GA.

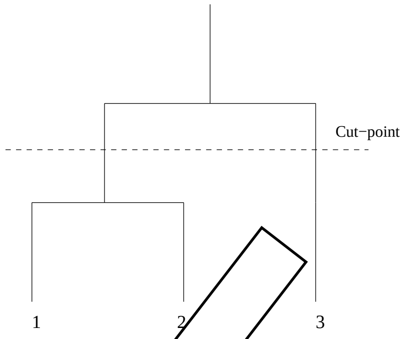


Figure 1. Hierarchical clustering dendrogram and cut-point.

2.1. Clustering

In this paper, the agglomerative-nesting (*agnes*) algorithm [15] was applied to build the initial set of *candidate libraries*. *Agnes* is an agglomerative, hierarchical clustering algorithm: it builds a hierarchy of clusters in such way that each level contains the same clusters as the first lower level, except for two clusters, which are joined to form a single cluster. In particular, agglomerative algorithms start building the dendrogram from the bottom of the hierarchy (where each one of the N entities represents a cluster), until at the $N - 1$ level all entities are grouped in a single cluster.

The key point of hierarchical clustering is determining the *cut-point*, i.e. the level to be considered in order to determine the actual clusters (e.g., in Figure 1 the cut-point determines a total of two clusters). As will be shown in Section 2.2, in this work such operation was supported by the Silhouette statistic.

2.2. Determining the Optimal Number of Clusters

To determine the optimal number of clusters, traditionally, people rely on the plot of an error measure representing the within cluster dispersion. The error measure decreases as the number of cluster k increases, but for some k the curve flattens. Traditionally, it is assumed that the error curve *elbow* indicates the appropriate number of clusters [11]. To overcome the limitation of such a heuristic approach, several methods have been proposed, see [11] for a comprehensive summary.

Kaufman and Russeeuw [15] proposed the *Silhouette* statistic for estimating and assessing the optimal number of clusters. For the observation i, let $a(i)$ be the average distance to the other points in its cluster, and $b(i)$ the average distance to points in the nearest cluster (but it own), then the Silhouette statistic is defined as:

DRAFT

$$s(i) = \frac{b(i) - a(i)}{max(a(i), b(i))} \qquad (1)$$

Kaufman and Russeeuw suggested choosing the optimal number of clusters as the value maximizing the average $s(i)$ over the dataset.

Notice that the Silhouette statistic, as most of the methods described in [11], has the disadvantage that it is undefined for one cluster, and thus it offers no indication of whether the current dataset already represents a good cluster. But, since our purpose is to split the original libraries into smaller ones, in our case this does not constitute a problem.

2.3. Genetic Algorithms

GA revealed their effectiveness in finding approximate solutions for problems where:

- The search space is large or complex;
- No mathematical analysis is available;
- Traditional search methods did not work; and, above all
- The problem is NP-complete [9, 27].

Roughly speaking, a GA may be defined as an iterative procedure that searches the best solution of a given problem among a constant-size population, represented by a finite string of symbols, the *genome*. The search is made starting from an initial population of individuals, often randomly generated. At each evolutionary step, individuals are evaluated using a *fitness function*. High-fitness individuals will have the highest probability to reproduce themselves.

The evolution (i.e., the generation of a new population) is made by means of two kind of operator: the *crossover operator* and the *mutation operator*. The crossover operator takes two individuals (the *parents*) of the old generation and exchanges parts of their genomes, producing one or more new individuals (the *offspring*). The mutation operator has been introduced to prevent convergence to local optima, in that it randomly modifies an individual's genome (e.g., flipping some of its bits if the genome is represented by a bit string). Crossover and mutation are respectively performed on each individual of the population with probability *pcross* and *pmut* respectively, where $pmut \ll pcross$.

The GA does not guarantee to converge: the termination condition is often specified as a maximal number of generations, or as a given value of the fitness function.

The GA behavior can be represented in pseudo-code as shown below:

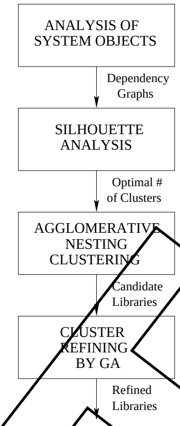


Figure 2. The re-factoring process.

```
Initialize population P[0];
g=0; // Generation counter
while(g < max_number_of_generations)

  //Apply the fitness function to the
  //current population
  Evaluate P[g];

  //Advance to the next generation
  g=g+1;

  //Make a list of pairs of individuals
  //likely to mate (best fitness)
  Select P[g] from P[g-1];

  //Crossover with probability
  //pcross on each pair
  Crossover P[g];

  //Mutation With probability
  //pmut on each individual
  Mutate P[g];
end while
```

3. The Re-factoring Method

This Section describes the proposed re-factoring process, derived from what already described in [4, 7], then refined with the Silhouette statistic for determining the optimal number of clusters, and with GA for minimizing inter-library dependencies. A flow diagram of the re-factoring process is depicted in Figure 2.

3.1. Basic Factoring Criteria and Representation

As described in [4, 7], given a system composed by m applications and n libraries, the idea is to re-factor the biggest libraries, splitting them in two or more smaller clusters, such that each cluster contains symbols used by a common subset of applications (i.e., we made the assumption that symbols often used together should be contained in the same library).

DRAFT

DRAFT

Given that, for each library l_j to be re-factored, a Boolean matrix MD, composed by $n + m$ rows and p_j columns, was built, such that:

$$MD[x,y] = \begin{cases} true & \begin{cases} x \leq m & object\ o_y\ is\ used \\ & by\ application\ a_x \\ & \\ x > m & object\ o_y\ is\ used \\ & by\ library\ l_{x-m} \end{cases} \\ & \\ false & otherwise \end{cases}$$

where $O \equiv \{o1, o2, \ldots, o_{p_j}\}$ is the set of objects of the library l_j (archiving p_j objects).

3.2. Determining the Optimal Number of Clusters

As explained in Section 2.2, the optimal number of clusters was computed on each MD matrix applying the Silhouette statistic. Giving the curve of the average Silhouette values for different numbers k of clusters, instead of considering the maximum (often too high for our re-factoring purpose), for some libraries we chose as optimal number the knee of that curve [15]. We also incorporated in the choice experts' knowledge, and we considered a tradeoff between excessive fragmentation and library size. Examples of Silhouette statistic are shown in Figure 4.

3.3. Determining the Sub-Optima Libraries by Clustering

Once known the number of clusters for each *"old library"*, agglomerative-nesting clustering was performed on each MD matrix. This builds a *dendrogram* and a vector of heights, that allow identifying k clusters. These clusters are the new *candidate libraries*.

A measure of the performances of the re-factoring process was introduced [7]. Let k the number of clusters $l_{x_1}, \ldots, l_{x_k}$ obtained from a library l_x. Then, the *Partitioning Ratio* PR_x can be defined as:

$$PR_x = 100 * \sum_{i=1}^{m} \frac{\sum_{j=1}^{k} |l_{x_j}| * d_{i,x_j}}{|l_x| * d_{i,x}} \tag{2}$$

where:

- $d_{i,x}$ is equal to one if the application i uses the library x (and zero otherwise);
- $|l_x|$ is the number of objects archived into library l_x.

The smaller is the PR, the most effective is the partitioning, in that the average number of objects linked (or loaded) from each application is smaller than using the old whole library.

3.4. Reducing Dependencies using Genetic Algorithms

The solution reached at the previous step presents a drawback: the number of dependencies between the new libraries could be high, forcing to load another library each time a symbol from that library is needed, therefore wasting the advantage of having new smaller libraries.

Of course, as shown in [7], an important step to perform is moving to dynamic-loadable libraries, so that each (small) library is loaded at run-time only when needed, and then unloaded when it is no longer useful. In this case, even if there are dependencies among libraries, the average number of libraries in memory is considerably smaller than in the original system. Given $O \equiv \{o_1, o2, \ldots, o_N\}$ the set of all N objects contained into the *candidate libraries* produced in the previous step, we built a dependency graph, defined as follows:

$$DG \equiv \{O, D\} \tag{3}$$

where O is the set of nodes representing the objects, and $D \subseteq O \times O$ the set of oriented edges $d_{i,j}$ representing dependencies between objects. Given two libraries, $L_a \subset O$ and $L_b \subset O$, we say that there is a dependency between the two libraries $\Leftrightarrow \exists\, i,j \mid o_i \in L_a,\ oj \in L_b,\ d_{i,j} \in D$.

The removal of inter-library dependencies can be therefore brought back to a graph partitioning problem that, as shown in [27], is NP-complete, and a GA was used to reach an approximate solution of the problem (i.e., minimize the number of dependencies).

A GA requires the specification of:

1. The genome encoding;
2. The initial population;
3. The fitness function;
4. The crossover operator;
5. The mutation operator; and
6. All GA parameters, such as the crossover and mutation probability, the population size and the number of generations.

An approach of clustering functions using GA was discussed in [8]. However, as shown below, in our case the genome encoding, the initial population, the mutation operator and the fitness function are different.

The encoding schema widely adopted in literature [8, 27] indicates each partition with an integer p such that $0 \leq p \leq k - 1$ (where k is the number of *candidate libraries*), and represents the genome as a N-size array G, where the integer p in position q means that the function q is contained into partition p.

However, as explained in [7], our purpose is the reduction of memory requirements for each application, therefore sometimes "cloning" an object in different libraries may help reducing the number of libraries to be linked by the application itself. The encoding schema mentioned above does not allow an object to be contained in more than one library.

We therefore adopted a bit-matrix encoding, where the genome g for each library to re-factor corresponds exactly to the MD matrix (where *true* values are indicated by "1" and *false* values by "0").

Clearly, the presence of the same object in more libraries is indicated by more "1" on the same column.

Instead of randomly generating the initial population (i.e., the initial libraries), the GA was initialized with the encoding of the set of libraries obtained in the previous step.

The fitness function was constructed to balance three factors:

1. The number of inter-library dependencies at a given generation;
2. The total number of objects linked to each application that, as said, should be as small as possible; and
3. The size of the new libraries.

Without taking into account the last item, it could happen that the GA, in the attempt to reduce dependencies, groups a large fraction of the objects in the same library, negatively affecting the PR.

The dependency graph DG was encoded as a matrix of adjacencies MDG:

$$MDG[x,y] = \begin{cases} 1 & o_x\ depends\ from\ o_y \\ 0 & otherwise \end{cases}$$

The first factor, the *Dependency Factor* ($DF(g)$) was defined as:

$$DF(g) = \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} MDG[i,j]\ neq(G[i], G[j]) \qquad (4)$$

where

$$neq(G[i], G[j]) = \begin{cases} 0 & G[i] = G[j] \\ 1 & G[i] \neq G[j] \end{cases}$$

The second factor, the *Linking Factor* ($LF(g)$) takes into account the number of objects linked by all the applications, if the *candidate libraries* were those produced at generation g. LF is obviously proportional to the PR, and therefore from this point on we will not distinguish between the two terms.

$LF(g)$ is computed as follows:

```
$MU = MD$
$LF(g) = 0$
$\forall\ application\ a_x$
    $\forall\ object\ o_y$
        $if\ MU[x,y] = true\ then$
            $find\ library\ l_k\ containing\ o_y$
            $LF(g) = LF(g) + |l_k|$
            $\forall\ o_j\ \in\ l_k$
                $MU[x,y] = false$
        $end\ if$
```

where $|l_k|$ is the number of objects contained in l_k. MU initially contains the matrix MD (see Section 3.1). As a library is linked, all its objects are removed from MU. It is worth noting that, if o_j is contained in more than one library, the "*most useful library*" (i.e., the library containing the largest number of objects needed by that application) is considered.

The third factor, the *Standard Deviation Factor* ($SF(g)$) can be thought of as the difference between the initial library sizes standard deviation and the actual (at the current generation) standard deviation. A similar factor was also applied in [27]. Given S_0 the array of library sizes for the initial population, and S_g the same for the g-th generation:

$$SF(g) = |\sigma_{S_0} - \sigma_{S_g}| \qquad (5)$$

Finally, the fitness function F was defined as:

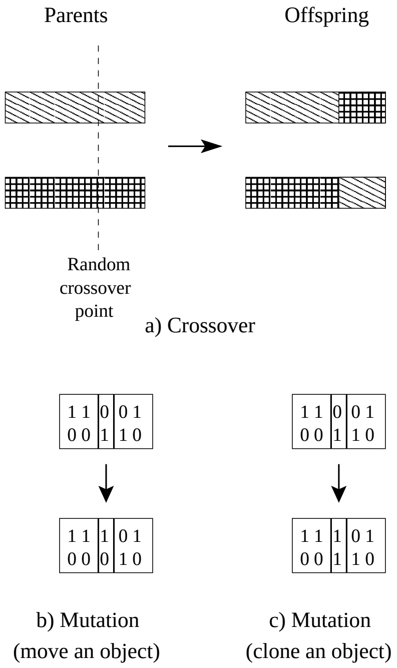
$$F(g) = \frac{1}{DF(g)\ +\ w_1\ LF(g)\ +\ w_2\ SF(g)} \qquad (6)$$

where w_1 and w_2 are real, positive weighting factor for the LF and SF contribution to the overall fitness function. The higher is w_1, the smaller will be the overall number of objects linked by applications; on the other hand, rising too much w_1 decreases dependency reduction.

Similarly, the higher is w_2, the more similar will be the result to the starting set of library, while an excessively higher w_2 could not allow a satisfactory dependency reduction. Finally, it is worth noting that our GA maximizes the fitness function, therefore F is the inverse of the weighted sum of the three factors.

As stated in (6), our fitness function is multi-objective [6, 12, 31]. Notice that we set a unitary weight to the DF, in that we aimed to maximize dependency reduction. Then we selected w_1 and w_2 using a trial-and-error, iterative procedure, adjusting them each time until the DF, LF and SF obtained at the final step were not satisfactory. The process was guided by computing each time the average values for DF, LF and SF, and also by plotting their evolution, in order to determine the 3D space region in which the population should evolve.

DRAFT



a) Crossover

b) Mutation (move an object)

c) Mutation (clone an object)

Figure 3. Genetic operators.

The crossover operator used in this paper is the *one point crossover*: given two matrices, both are cut at the same random column, and the two portions are exchanged (Figure 3a). The mutation operator works in two modes:

1. Normally, it takes a random column and randomly swaps two rows: this means that, if the two swapped bits are different then an object is moved from a library to another (Figure 3b);

2. With probability $pclone < pmut$, it takes a random position in the matrix: if it is zero and the library is dependent on it, then the mutation operator clones the object into the current library (Figure 3c).

Of course the cloning of an object increases both LF and SF, therefore it should be minimized. Our GA activates the cloning only for the final part of the evolution (after 66% of generations in our case studies).

Our strategy favors dependency minimization by moving objects between libraries; then, at the end, we attempt to remove remaining dependencies by cloning objects.

The population size and the number of generations were chosen by an iterative procedure, doubling both each time until the obtained DF and LF (and thus the PR) were equal to those at the previous step.

3.5. Tool Support

To support the re-factoring process, different tools were needed, some of which already described in [7]. In particular:

- *The application identifier* that, using the `nm` Unix tool, identifies the list of object modules containing the `main` symbol;

- *The dependency graph extractor*, also based on the `nm tool`, that produces the MD and the MDG matrices. The tool presented in [7] was modified in order to produce information in the format required by our GA tool;

- *The number of clusters identifier:* as said in Section 2.1, the number of clusters was determined using the Silhouette statistic. In particular, implementations available in the *cluster* package of the *R Statistical Environment* [1, 13] were used;

- *The library re-factoring tool*: it supports the process of splitting libraries in smaller clusters. As said in Section 3.3, this is performed by clustering algorithms. Again, the cluster analysis is performed by the *agnes* function available under the *cluster* package of the *R Statistical Environment*; and

- *The GA library refiner:* implemented in C++ using the *GAlib* [30].

4. Case Studies

Although our primary objective was to improve the re-factoring of *GRASS* biggest libraries, we applied our process to other open source systems, different for purpose and size. This gave us more empirical evidence for the validity of the proposed approach. As explained in Section 5, for each system we chose to re-factor the biggest libraries. Characteristics of the four systems analyzed are shown in Table 1.

System	Ver	KLOC	Apps	Libs	Libs to re-factor
GRASS	Snap. 02-22-2002	1,014	517	43	3
MySQL	3.23.51	478	38	24	2
Samba	2.2.5	293	16	2	2
KDE-QT	3.0.3	4,196	589	456	4

Table 1. Case studies characteristics.

4.1. GRASS

GRASS (Geographic Resources Analysis Support System, `http://grass.itc.it`) is an Open Source,

DRAFT

raster/vector Geographical Information System (GIS), having integrated image processing and data visualization subsystems [23]. Supported platforms at the date of writing comprise Linux/PC, SUN, HP/UX, MacOSX, MS-Windows/Cygwin, iPAQ/Linux and others.

GRASS modules (commands) are invoked within a shell environment (also the current graphical user interface runs commands within a shell). The *GRASS* parser is a collection of subroutines allowing the programmer to define options (parameters) and flags that make up the valid command line input of a *GRASS* command.

GRASS provides an ANSI C language API with several hundreds of GIS functions which are utilized in the *GRASS* modules, from reading and writing maps to area and distance calculations for georeferenced data as well as attribute handling and map visualization. Details of *GRASS* programming are covered in the "*GRASS* 5.0 Programmer's Manual" [22].

4.2. Samba

Samba (`http://www.samba.org`) is a freely available file server that runs on Unix and other operating systems (usually to share resources between Unix-based systems and Microsoft-based systems). The code has been written to be as portable as possible. It has been "ported" to many unixes (Linux, SunOS, Solaris, SVR4, Ultrix, etc.).

Samba consists of two key programs, plus a bunch of other utilities. The two key programs are `smbd` and `nmbd`. They implement four basic services: file sharing & print services, authentication and authorization, name resolution and service announcement (browsing). Moreover, *Samba* comes with a variety of utilities. The most commonly used utilities are: `smbclient` a simple SMB (Server Message Block, a protocol for sharing general communications abstractions such as files, printers, etc.) client; `nmblookup` a NetBIOS name service client, and `swat` which is the *Samba Web Administration Tool*, it allows the configuration of Samba remotely, using a web browser.

4.3. MySQL

MySQL (`http://www.mysql.com/`) is an open source, fast, multi-threaded, multi-user SQL database server, intended for mission-critical, heavy loaded production systems. *MySQL* is written using both C and C++ (the latter constitutes no big deal for our approach, since dependency graph were extracted from object files), and can be compiled with several different C/C++ compilers.

The power of *MySQL* is in its fastness: in order to pursue this objective, some advanced features (e.g., nested queries) are not available, while others (e.g., transactions) were introduced only in the latest version of the database server.

4.4. KDE-QT

KDE is an open source desktop environment for Unix workstations. It was developed to facilitate Unix desktop interaction and programming, in a way more similar to MacOS or MS-Windows. In particular, *KDE* provides some features not available under X11, such as common Drag and Drop protocol, desktop configuration, unified help system, application development framework, consistent look-and-feel and menu system, and internationalization.

The *KDE* distribution consists of 19 packages: the base package (*KDE-Base*), the library package (*KDE-Libs*), the development package (*KDevelop*), the network package (*KDE-Network*), the graphics package (*KDE-Graphics*), the multimedia package (*KDE-Multimedia*), and others (see `http://www.kde.org` for further details).

KDE was developed using a multi-platform C++ GUI application framework, *QT* (`http://www.trolltech.com/products/qt/`). As shown in Section 5, *QT* library is quite big, therefore its re-factoring could be useful for porting graphical applications on hand-held devices.

5. Case Studies Results

This section reports results obtained applying the proposed re-factoring process on the different systems described in Section 4. Table 2 reports results of the re-factoring process applied on selected libraries of the systems analyzed. The table reports, for each library:

- The number of objects composing the library;
- The number of *candidate libraries* the original library is re-factored into, and the corresponding Silhouette statistic value;
- The number of inter-library dependencies and the PR before applying the GA; and
- The number of inter-library dependencies and the PR after applying the GA.

GA parameters were fixed, after a proper calibration, as follows: $pcross = 0.8$, $pmut = 0.2$, $pclone = 0.1$, $population\ size = 300$. The number of generations required varied from 1500 to 3000.

As shown in Figure 4, different heuristics were followed to choose the optimal number of clusters, such as the curve knee (`libmsqlclient` and KDE-`libkio`) or the maximum (*Samba* and *GRASS*-`libgis`). A similar approach was followed for all others libraries.

All $PR < 1/k$ shown in Table 2 are due to the presence of objects unused by the current set of applications, therefore clustered in a separate library.

DRAFT

System	Library	# of objects	Candidate Libraries (k)	Silhouette statistic	Before GA		After GA	
					DF	PR	DF	PR
GRASS	`libgis`	184	4	0.70	356	10%	13	8%
	`libdbmi`	97	3	0.78	237	31%	5	23%
	`libvect`	54	3	0.57	66	46%	4	25%
Samba	`lib + libsmb`	77	2	0.72	106	72%	2	64%
MySQL	`libmsqlclient`	80	3	0.53	187	76%	7	70%
	`libmysys`	92	2	0.53	158	89%	1	66%
KDE-QT	`lib-qt`	403	5	0.79	147	11%	9	5%
	`libkhtml`	100	3	0.58	0	83%	0	26%
	`libkio`	129	4	0.56	2	16%	0	7%
	`libmpeg`	138	3	0.74	68	48%	1	17%

Table 2. Results of the re-factoring process.

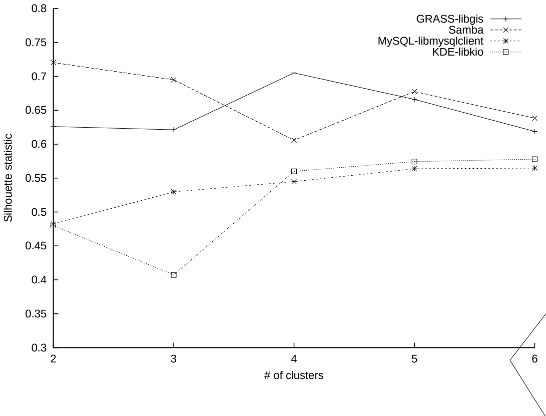


Figure 4. Examples of Silhouette statistic behaviors.

The biggest *GRASS* library to re-factor was `libgis`, composed by 184 objects. The library was split in only four clusters (according to the Silhouette statistic) instead of the six proposed in [7]. The GA reduced dependencies from 356 to 13 keeping the PR almost constant (from 10% to 8%).

`libdbmi` and `libvect` were both re-factored in three clusters: in the first case the three cluster structure (also suggested by developers) reflected, as explained in [7], the separation of high-level functionalities from low-level functionalities and unused objects. GA allowed, for both libraries, a considerable reduction of inter-library dependencies (from 237 to 5 for `libdbmi` and from 66 to 4 for `libvect`) also slightly reducing the PR.

All the new *GRASS* libraries received a positive feedback by original developers, indicating us the effective and useful re-factoring.

The re-factoring process performed for *Samba* was quite different from all others: instead of re-factoring big libraries, we tried to re-organize the two existing libraries, in order to minimize dependencies (initially 106) between them. The resulted new libraries exhibited only two dependencies (then easily manually removed), and a PR (64%) smaller than the original (72%).

The two biggest *MySQL* libraries (`libmsqlclient` and `libmysys`) were re-factored in three and two clusters, and the GA allowed us to minimize dependencies (1 and 7).

`libqt` represented for us an interesting challenge, it that it was really a big library (403 objects) and it was used by a large number (630) of *KDE* applications and other libraries. According to the Silhouette statistic, it was split into five clusters; then, the GA allowed to decrease the number of inter-library dependencies from 147 to 9 and the PR from 11% to 5%.

When re-factoring `libmpeg`, instead of considering the use made by all *KDE* applications, only applications contained in the *kde-multimedia* package (the only ones using that library) were taken into account. Also in this case, the dependencies and PR reduction was successfully performed.

The purpose of applying GA was slightly different for `libkhtml` and `libkio`: the number of inter-library dependencies was zero for the former, and very small (2) for the latter. Instead, GA allowed a considerable reduction of PR (from 83% to 26% and from 16% to 7%). The high PR reduction for `libkhtml` was due to the fact that, in performing agglomerative clustering, some objects used by applications were clustered together with a large number of unused objects, that were linked by all applications needed by the former objects. In this case, GA allowed grouping in a separate cluster only the unused objects.

6. Related Work

Literature reports several works applying clustering or concept analysis (CA) to software system modules clustering and/or restructuring, identifying objects, and recovering or building libraries.

An overview of CA applied to software reengineering problems was shown by G. Snelting in his seminal work [26], where he used CA in several re-modularization problems such as exploring configuration spaces (see also [17]), transforming class hierarchies, and re-modularizing COBOL systems. A comparison between clustering and CA was presented in [18]. We share with them the idea to apply an agglomerative-nesting clustering to a Boolean usage matrix, although in [18] the matrix indicated the uses of variables by programs.

A survey of clustering techniques applied on software engineering was presented by Tzerpos and Holt in [29]. The same authors presented in [28] a metric to evaluate the similarity of different decompositions of software systems. Applications of clustering to re-engineering can be found in [3] and [21]. In [3] a method for decomposing complex software systems into independent subsystems was proposed by Anquetil and Lethbridge. Merlo et al. [21] exploited comments, as well as variable and function names, to cluster files. Our work shares with [20] the idea of analyzing intra-module and inter-module dependency graphs, finding a tradeoff between having highly cohesive libraries and a low inter-connectivity.

GA has been recently applied in different fields of computer science and software engineering. An approach for partitioning a graph using GA was discussed in [27]. Similar approaches were also shown in [25, 5, 24]. Maini et al. [19] discussed a method to introduce the problem knowledge in a non-uniform crossover operator, and presented some examples of its application (also a graph partitioning problem). We share with this work the idea of using operators incorporating the problem knowledge: in our case the mutation operator, as discussed in Section 3, clones objects if, after a given percentage of generations, inter-library dependencies are still present. GA were used by Doval et al. [8] for identifying clusters on software systems. We share with this paper the idea of a software clustering approach using GA, trying to minimize inter-cluster dependencies.

In [4] we proposed the idea of recovering libraries and creating a source file directory structure using CA. This paper shares with [4] the idea of finding libraries searching for sets of objects used by common groups of applications. The re-factoring of *GRASS* was proposed in [7], where several activities were carried out in order to re-factoring *GRASS* libraries. In particular, unused symbols were identified and pruned, clones were re-factored, and a preliminary work aimed at splitting the biggest libraries in clusters was performed.

As stated in the introduction, this paper aims to refine the library re-factoring approach, determining the optimal number of clusters with the Silhouette statistics and minimizing the number of inter-cluster dependencies using GA.

7. Conclusions

The proposed re-factoring process allowed obtaining smallest, loosely coupled libraries from the original biggest ones. In particular, the Silhouette statistic gave us information that, together with the experts' knowledge, allowed defining the optimal number of new *candidate libraries*. Then, the GA, initialized from the clusters produced by agglomerative-hierarchical clustering, significantly reduced the number of inter-dependencies, keeping lower, at the same time, the ratio between the average number of objects linked by each application, and the number of object linked before re-factoring (Partitioning Ratio).

The method was successfully applied to our main case study (*GRASS*), as well as to other system like *KDE*, where the size of some libraries (especially `libqt`, composed by 319 objects) was really considerable.

Our approach is language-independent since information is gathered from object modules, thus it could be applied to object code produced by any known programming language.

Work in progress is devoted to incorporate experts' knowledge into genetic algorithms, in order to cluster objects taking into account not only the use made by applications, but also trying to cluster objects having similar purpose.

8. Acknowledgments

We are grateful to the *GRASS* development team for the support, the information provided, and the feedback on the re-factored artifacts.

Giuliano Antoniol and Massimiliano Di Penta were partially supported by the project "Software Architectures for Heterogeneous Access Networks Infrastructures - SAHARA" funded by "Ministero dell'Istruzione, dell'Università e della Ricerca Scientifica- MIUR".

References

[1] The R project for statistical computing. `http://www.r-project.org`.

[2] M. R. Anderberg. *Cluster Analysis for Applications*. Academic Press Inc., 1973.

[3] N. Anquetil and T. Lethbridge. Extracting concepts from file names; a new file clustering criterion. In *Proceedings of the International Conference on Software Engineering*, pages 84–93, April 1998.

DRAFT

[4] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. A method to re-organize legacy systems via concept analysis. In *Proceedings of the IEEE International Workshop on Program Comprehension*, pages 281–290, Toronto, ON, Canada, May 2001. IEEE Press.

[5] T. N. Bui and B. R. Moon. Genetic algorithm and graph partitioning. *IEEE Transactions on Computers*, 45(7):841–855, Jul 1996.

[6] K. Deb. Multi-objective genetic algorithms: Problem difficulties and construction of test problems. *Evolutionary Computation*, 7(3):205–230, 1999.

[7] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo. Knowledge-based library re-factoring for an open source project. In *Proceedings of IEEE Working Conference on Reverse Engineering*, Richmond - VA, Oct 2002 (to appear).

[8] D. Doval, S. Mancoridis, and B. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Software Technology and Engineering Practice (STEP)*, pages 73–91, Pittsburgh, PA, 1999.

[9] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[10] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub Co, Jan 1989.

[11] A. Gordon. *Classification - (2nd edition)*. Chapman and Hall, London, 1988.

[12] J. Horn, N. Nafpliotis, and D. E. Goldberg. A Niched Pareto Genetic Algorithm for Multiobjective Optimization. In *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, volume 1, pages 82–87, Piscataway, New Jersey, 1994. IEEE Service Center.

[13] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[14] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[15] L. Kaufman and P. Rousseeew. *Finding groups in data: an introduction to cluster analysis*. Wiley-Inter Science, Wiley - NY, 1990.

[16] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.

[17] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proc. of the 16th International Conference on Software Engineering*, pages 49–57, Sorrento Italy, May 1994.

[18] T. Kuipers and A. van Deursen. Identifying objects using cluster and concept analysis. In *Proceedings of the International Conference on Software Engineering*, pages 246–255, June 1999.

[19] H. Maini, K. Mehrotra, C. Mohan, and S. Ranka. Knowledge-based nonuniform crossover. In *IEEE World Congress on Computational Intelligence*, pages 22–27, 1994.

[20] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IEEE Proceedings of the 1998 Int. Workshop on Program Comprehension (IWPC'98)*, 1998.

[21] E. Merlo, I. McAdam, and R. De Mori. Source code informal information analysis using connectionist model. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1339–44, Load Altos Calif, 1993.

[22] M. Neteler, editor. *GRASS 5.0 Programmer's Manual. Geographic Resources Analysis Support System*. ITC-irst, Italy, `http://grass.itc.it/grassdevel.html`, 2001.

[23] M. Neteler and H. Mitasova. *Open Source GIS: A GRASS GIS Approach*. Kluwer Academic Publishers, Boston/U.S.A; Dordrecht/Holland; London/U.K., 2002.

[24] B. Oommen and E. de St. Croix. Graph partitioning using learning automata. *IEEE Transactions on Computers*, 45(2):195–208, Feb 1996.

[25] S. Shazely, H. Baraka, and A. Abdel-Wahab. Solving graph partitioning problem using genetic algorithms. In *Midwest Symposium on Circuits and Systems*, pages 302–305, 1998.

[26] G. Snelting. Software reengineering based on concept lattices. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 3–10, March 2000.

[27] E. Talbi and P. Bessire. A parallel genetic algorithm for the graph partitioning problem. In *ACM International Conference on Supercomputing*, Cologne, Germany, 1991.

[28] V. Tzerpos and R. C. Holt. MoJo: A distance metric for software clusterings. pages 187–195.

[29] V. Tzerpos and R. C. Holt. Software botryology: Automatic clustering of software systems. In *DEXA Workshop*, pages 811–818, 1998.

[30] M. Wall. GAlib - a C++ library of genetic algorithm components. `http://lancet.mit.edu/ga/`.

[31] E. Zitzler, K. Deb, and L. Thiele. Comparison of Multiobjective Evolutionary Algorithms on Test Functions of Different Difficulty. In A. S. Wu, editor, *Proceedings of the 1999 Genetic and Evolutionary Computation Conference. Workshop Program*, pages 121–122, Orlando, Florida, 1999.

DRAFT