Using Genetic Algorithms to Solve Software Clustering Problem

Violeta T. Bojikova[1] and Milena M. Karova[2]

Abstract - **In this paper is presented a genetic algorithm (GA), which solves software clustering problem, using a fitness function that is based on maximizing the cohesiveness of clusters and minimizing inter-cluster coupling. Software clustering facilitates program understanding and reengineering. The presented approach differs strongly from other published genetic approaches because of the choice of the fitness function, genetic operator's realization and the presence of determinism.**

The goal of the developed genetic project is to produce better clustering results, with regard to the solution quality and the algorithm complexity. s.

Keywords – **search-based software clustering algorithms, software clustering, genetic algorithms, program reengineering**

I. INTRODUCTION

Most interesting software systems are large and complex, and as a consequence, understanding the structure of these systems is difficult [1,2,3]. Software Engineering books advocate the use of documentation as an essential tool for describing a system's intended behavior, and for capturing the system's structure. In practice, however, we often find that accurate and current design documentation does not exist. This problem is exacerbated because the original designers and developers of the system are often no longer available for consultation.

In the absence of advice or documentation about a system's structure, software maintainers are left with several choices. First, they can manually inspect the source code to develop a mental model of the system organization. This approach is often not practical because of the large number of dependencies between the source code components. Another alternative that is now becoming available to software maintainers is to use automated tools to produce useful information about the system structure. A primary goal of these tools is to analyze the low-level dependencies in the source code, and cluster them into meaningful subsystems.

A subsystem is a collection of source code resources that closely collaborate with each other to implement a high-level feature, or provide a high-level service to the rest of the system. Typical source code resources that are found in subsystems include modules, classes, packages, files. Subsystems facilitate program understanding by logically treating many low-level source code resources as a single high-level entity. Subsystems are not specified in the source code explicitly, as most programming languages do not support this kind of structuring yet. Because source code lacks the descriptiveness necessary to specify subsystems, we can use tools for automatically recovering the subsystems of a software system from the source code. Software designers use directed graphs to represent the structure of a software system and to make the structure of complex software systems more understandable.

The graph is formed by representing the modules of the system as nodes, and the dependencies between the modules as edges. We refer to this graph as the Module Dependency Graph (MDG) of a software system. Each partition of the MDG consists of a set of non-overlapping clusters that cover all of the nodes of the graph. The goal is to partition the MDG into meaningful subsystems.

Given an MDG with n components that are partitioned into k distinct clusters (subsystems), the number of ways to cluster the MDG grows exponentially with respect to the number of modules in the system. The general problem of clustering software systems is NP hard, however, the formal proof of this remains an open problem. Most researchers have addressed the software clustering problem by using heuristics to reduce the execution complexity to a polynomial upper bound.

Our approach treats clustering as an optimization problem, where the goal is to find a good (possibly optimal) partition. To explore the extraordinarily large solution space of all possible partitions for a given MDG, we use a Genetic Algorithm (GA).

II. GENETIC ALGORITHMS: ENCODING, OBJECTIVE FUNCTION, OPERATORS

Genetic algorithms apply ideas from the theory of natural selection to navigate through large search spaces efficiently [2]. GAs perform surprisingly well in highly constrained problems, where the number of "good" solutions is very small relative to the size of the search space.

GAs operate on a set (*population*) of strings (*individuals*), where each string is an encoding of the problem's input data. The number of strings in a population is defined by the *population size*. The larger the population size, the better the chance that an optimal solution will be found. Since GAs are very computationally intensive, a trade-off must be made between population size and execution performance. In the

[1] Violeta Bojikova is with the Department of Computer Science Varna Technical University, Bulgaria, e-mail: vbojikova2000@yahoo.com

[2] Milena Karova is with the Department of Computer Science Varna Technical University, Bulgaria, e-mail:mkarova@ieee.bg

case the population size is defined by the number of sequential algorithms that form the stating population [4,5].

Each string's *fitness* is calculated using an objective function. In each generation, a new population is created by taking advantage of the fittest individuals of the previous generation.

Genetic algorithms are characterized by attributes such as objective function, encoding of the input data, genetic operators, and population size. After describing these attributes, we describe the GA algorithm in Section III.

- The objective function

The objective function k is used to assign a fitness value to each individual in the population. In the case, it is designed so that an individual with a lower objective function value have a high fitness and represents a better solution to the problem.

The basic idea of this objective function is "well designed software systems are organized into cohesive clusters that are loosely interconnected".

The weight – Wk of each cluster $g_i \in$ MDG with $x_i \in X$ components (nodes) corresponds to the restrictive condition Wo, where w_i – is the label of node - x_i and presents the number of node's elements (i.e. modules):

The value of "k", where k_{ij} is the number of inter-edges (*i.e.*, external edges that cross cluster boundaries) between nodes x_i and x_j is calculated as follow:

$$k = \sum_{i=1..M} \sum_{j=1..M} k_{ij}, \forall i \neq j$$

M is the number of clusters in the current partition of the MDG.

- Encoding

GAs operate on an *encoding* of the problem's input data. The choice of the encoding is extremely important for the execution performance of the algorithm. A poor encoding can lead to long-running searches that do not produce good results.

Each node in the graph G=(X,U), where N=|X| has a unique numerical identifier assigned to it (*e.g.*, node x1 is assigned the unique identifier 1, node two is assigned the unique identifier 2, and so on). These unique identifiers define which position in the encoded string S will be used to define that node's cluster. We can use the next encoding string S:

$S=s_1\ s_2\ s_3\ \ s_N$, where $s_i \in \{1\ldots M\}$, $i \in \{1\ldots N\}$.

Therefore, the first character in the string S - s_1, indicates that the first node (x1) is contained in the cluster labeled s_1. Likewise, the second node (x2) is contained in the cluster labeled s_2, and so on.

- Genetic operators

GAs feature the following three basic operators, which are executed sequentially by the GA:

1. Selection
2. Crossover
3. Mutation

During **selection**, pairs of individuals are chosen from the population according to their fitness. We use in GA *competition selection*. The 2 individuals with the high fitness are selected from the old population to be included in the new population. This selection is complemented with *elitism*. Elitism guarantees that the fittest individual of the current population is copied to the new population.

Crossover is performed immediately after selection. The crossover operator is used to combine the pairs of selected strings (parents) to create new strings that potentially have a higher fitness than either of their parents.

During crossover, each pair of strings is split at a random position. Two new strings are then created by swapping these random characters of the selected individuals. Thus, two strings are used to create two new strings, maintaining the total population of a generation constant.

The **mutation** operator is applied if the strings resulting from the crossover process are identical. The mutation is applied over one of the string – random exchange of random number of nodes between the clusters of one of the solutions (one of the individuals).

After that, we can apply **restoring** – to satisfy the restrictive condition Wo.

III. THE GA ALGORITHM

GAs use the operators defined above to operate on the population through an iterative process, which is as follows:

1. Generate the initial population, creating a set of constructive sequential solutions (strings) of fixed size [4,5].
2. Create a new population by applying the selection operator to select pairs of strings.
3. Apply the crossover operator to the pairs of strings of the new population.
4. Apply the mutation operator to each string in the new population.
5. Apply the restoring operator, if the new strings are identical.
6. Replace the old population with the newly created population.
7. If the number of iterations is less than the maximum and the population is changed, go to step 2. Else, stop the process and display the best answer found.

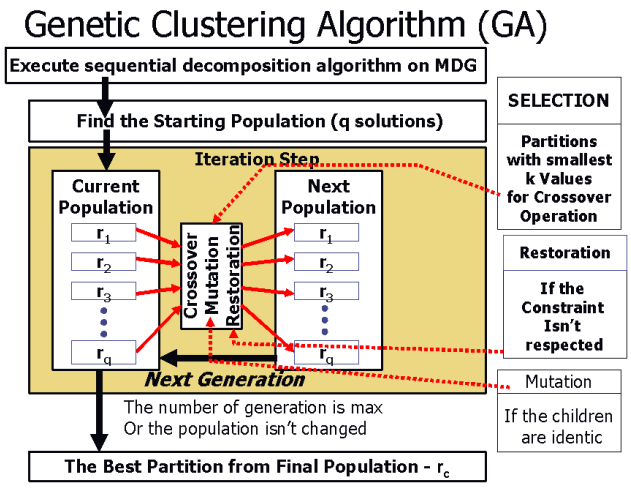

fig.1. Genetc Clustering algorithm

IV. CONCLUSION

In this paper we describe a Genetic Algorithm (fig.1) for software clustering. Over the past few years we have seen an increasing interest and activity in software clustering research [1,2,3]. Clustering has been applied in the fields of mathematics, social sciences, engineering and biology for many years. However, only within the last 25 years have researchers investigated and applied clustering techniques to the software domain. This work has resulted in new clustering techniques, as well as the adaptation of classical clustering techniques to the particularities of the software domain.

Improving Optimization Techniques is one of the new research directions in the area of software clustering. Genetic algorithms have been shown to produce good results in optimizing large problems. Although the initial results with the genetic realization in [2] are promising, additional study is needed. We are presented an GA algorithm that has an other goal function and apply alternative selection and mutation operators. Additional research is needed to proof the effectiveness of this algorithm.

REFERENCES

[1]. Comparing the Decompositions Produced by Software Clustering Algorithms using Similarity Measurements, Spiros Mancoridis and Brian Mitchell
IEEE Proceedings of the 2001 International Conference on Software Maintenance (ICSM'01). IEEE.

[2]. Spiros Mancoridis, Brian Mitchell, and Diego Doval, Automatic Clustering of Software Systems using a Genetic Algorigthm, , IEEE Proceedings of the 1999 International Conference on Software Tools and Engineering Practice (STEP'99)

[3]. Search Based Reverse Engineering", by B. S. Mitchell, S. Mancoridis, M. Traverso. In the ACM *Proceedings of the 2002 International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, Ischia, Italy, July, 2002. pp. 431-438.

[4]. Software Architecture Decomposition, Violeta Bojikova, M. Mitev, Proceedings of the 14th Int'l Conference SAER'2000, Varna, 2000, pp. 173-177.

[5]. An Approach to measure the Cost of Program restructuring, Violeta Bojikova, M. Karova, Proceedings of papers, Volume 2, pp 669-671, 2002, Nish, Yugoslavia