

# A New Encoding Scheme and a Framework to Investigate Genetic Clustering Algorithms

Saeed Parsa and Omid Bushehrian

Iran University of Science and Technology

Email: [parsa@iust.ac.ir](mailto:parsa@iust.ac.ir)

Email: [bushehrian@iust.ac.ir](mailto:bushehrian@iust.ac.ir)

*In this paper a new encoding scheme and a software environment, called DAGC, to develop and evaluate genetic clustering algorithms is described. DAGC facilitates experiments with genetic clustering algorithms by providing an extensible library of components to assemble new algorithms or modify existing ones. The algorithms may be executed within the environment on caterpillar or random graphs or class dependency graphs extracted from a given source code. The resultant clustering can be stored in a database, for later analysis. DAGC allows confidence analysis by automatically deriving a consolidated model from different clustering results for a given graph. We also offer a new clustering algorithm, called DAGC. The results of comparing the DAGC algorithm with a well known algorithm, Bunch, are presented.*

*ACM Classification: D.2.7 (Software Engineering – Distribution, Maintenance, and Enhancement)*

## 1. INTRODUCTION

Most interesting software systems are large and complex and, as a consequence, understanding their structure is difficult. Clustering is a key activity in reverse engineering to discover a better design of the systems or to extract significant concepts from the code (Mitchell and Mancoridis, 1999; Davey and Burd, 2000; Anquetil and Lethbridge, 1999; Mancoridis *et al*, 1998). Hierarchical approaches to clustering seem to be useful for program understanding and knowledge discovery, because in general they allow the original problem to be studied at different levels of detail by navigating up and down the hierarchy. However, it is a difficult problem to find the appropriate height at which to prune a hierarchy of clusters to obtain an optimal partitioning (Anquetil and Lethbridge, 1999; Mancoridis *et al*, 1998).

A key activity in reverse engineering consists of gathering the software entities (modules, routines, etc.) that compose the system into meaningful (highly cohesive) and independent (loosely coupled) groups. This is an optimization activity that can be best carried out by optimization clustering techniques (De Jong and Spears, 1989). Optimization clustering algorithms are NP-Hard (De Jong and Spears, 1989; Garey and Johnson, 1979). Genetic algorithms are widely believed to be effective on NP-Hard global optimization problems, such as clustering, and they can provide good sub-optimal solutions in reasonable time (De Jong and Spears, 1989).

Genetic clustering algorithms are very subjective (Mitchell and Mancoridis, 2001a). They may perform well for a certain type of system while producing unacceptable decompositions for the other systems. Moreover, the literature often does not describe the types of systems for which a

---

*Copyright© 2005, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the JRPIT copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.*

*Manuscript received: 29 December 2003*

Communicating Editor: John Yearwood

certain clustering algorithm does not perform well. Hence, there is a need for software environments to facilitate creation, modification and evaluation of these algorithms. To achieve this, well known frameworks such as Bunch (Mitchell and Mancoridis, 1999) and CRAFT (Mitchell and Mancoridis, 2001b) have provided environments to run and view the results of clustering on software systems. CRAFT is capable of accepting and evaluating new algorithms. GAME (Gokel *et al*, 1997) is an environment with a fixed set of components to assemble genetic algorithms for circuit design. In Moon (1994) a framework for interactive modification of objective functions for genetic clustering algorithms is presented. Within this environment, to apply various constraints on clustering, the objective function can be easily modified and the resultant clustering may be visualized.

There is a great variety of generic class libraries for optimization problems. A main reason for this variety, is that most of these libraries are hard to use or too restrictive or just plain bad products (Keijzer *et al*, 2002). For instance, GALIB (Wall, 1995) is a widely used library which lacks flexibility in many areas (Heitkoter and Beasley, 1995); EOLib is a generic class library for heuristic search (Woodruff, 2002). It should be noted that even if there are reusable software components for Genetic Algorithms (GA), the success of respective implementations still depends on appropriate definitions of the concepts for the specific problem. That is, reusable components support the development of specific genetic algorithms, where the success generally depends on the meaningful definition of several concerns and the efficient implementation of the problem-specific parts.

Our objective has been to develop an environment to experiment with the effects of applying different schemes for the components of genetic clustering algorithms for software re-modularization. To achieve this, we have developed a flexible software environment for Dynamic Assembly of Genetic Clustering components (DAGC).

In DAGC, clustering algorithms may be assembled (created) or modified by selecting the parts from a library of components, which can be modified by the user. Here, a component is a function with a standard interface, representing a genetic operator such as crossover or any other part of a genetic clustering algorithm, such as random number generators or replacement schemes.

Another major issue concerning genetic clustering algorithms is the way that each solution to the problem is represented as a chromosome. This is called the encoding scheme. Encoding schemes affect performance of genetic algorithms because a good encoding reduces the search space of the GA. In this paper a permutation based encoding is presented. This permutation based encoding maps each cluster to a permutation of  $n$  numbers ( $n$  is the number of clusters) so the search space has been reduced. Permutation based encodings use a special class of crossover operators called order based crossover which is explained in Section 6.1.

The structure of the rest of this paper is as follows: In Section 2, software modularization process is described. Section 3 presents an overview of genetic algorithms and a general clustering algorithm. The algorithm makes use of a set of components, which are described in Section 4. Section 4 describes the components used as building blocks for the general algorithm. The overall architecture of the DAGC framework is described in Section 5. In Section 6 the evolution and evaluation of a new clustering algorithm developed within the DAGC is described. This algorithm uses our new encoding and crossover operators. Encoding schemes affect performance of genetic algorithms because a good encoding reduces the search space of the GA. In this paper a new permutation based encoding scheme is presented. Permutation based encodings use a special class of crossover operators called order based crossover which is explained in Section 6.1. Finally in Section 7, the DAGC facilities for evaluation of clustering results are described.

## 2. SOFTWARE RE-MODULARIZATION

Software Systems contain a finite set of software components along with a finite set of relationships that govern how the software components interact with each other. In this paper, a software component refers to a number of interrelated classes. The goal of the software modularization process is to automatically partition the components of a system into clusters (subsystems) so that the resultant organization concurrently minimizes inter-connectivity (i.e, connections between the components of two distinct clusters) while maximizing intra-connectivity (i.e, connections between the components of the same cluster)(Abreu *et al*, 2000; Davey and Burd, 2000; Anquetil and Lethbridge, 1999; Mancoridis *et al*, 1998; Saeed *et al*, 2003).

We accomplish this task by treating clustering as an optimization problem where our goal is to minimize an objective function based on a formal characterization of the trade-off between inter and intra connectivity of the clusters. The clusters, once discovered, represent high level components abstraction of a system's organization.

A fundamental assumption underlying our approach is that well-designed software systems are organized into cohesive clusters that are loosely inter-connected (Anquetil and Lethbridge, 1999; Wiggerts, 1997).

A typical software remodularization process is illustrated in Figure 1. The process begins with parsing the source code of the software, to be re-modularized. While parsing, the most relevant information for constructing a call graph are extracted from the source code and described in an abstract form (Abreu *et al*, 2000). The call graph is then extracted and clustered using software modularization criteria (Mitchell, 2002).

## 3. GENETIC CLUSTERING

The idea has been to develop a software environment to facilitate researchers' investigations on development of optimal genetic clustering algorithms for automatic modularization of software systems. To achieve this goal, a comprehensive study of the existing algorithms was carried out. We arrived at the results listed below.

1. There are a fixed set of component types, mostly used in these algorithms.
2. For each component type, as described in Section 4, a standard interface could be defined.
3. Using standard interfaces, various schemes may be applied to define the components without any need to change the existing source code for the algorithm.
4. For each component, a number of implementations could be provided.
5. A general clustering algorithm, calling the components could be developed.
6. New algorithms could be created by selecting the components, using the interface illustrated in Figure 1.

Genetic algorithms are a class of powerful, robust search techniques based on genetic inheritance and the Darwinian metaphor of "Natural Selection". These algorithms maintain a finite memory of individual points on the search landscape known as the "population". Members of the population are usually represented as strings written over some fixed alphabet, each of which has a scalar value attached to it reflecting its quality or "fitness". The search may be seen as the iterative application of a number of operators, such as selection, recombination and mutation, to the population with the aim of producing progressively fitter individuals.

In general, there are a fixed set of operators and components appearing in genetic clustering algorithms which are highlighted in bold in Figure 2. In Figure 2, a pseudo code description of a generalized genetic clustering algorithm, used within the DAGC, is presented.

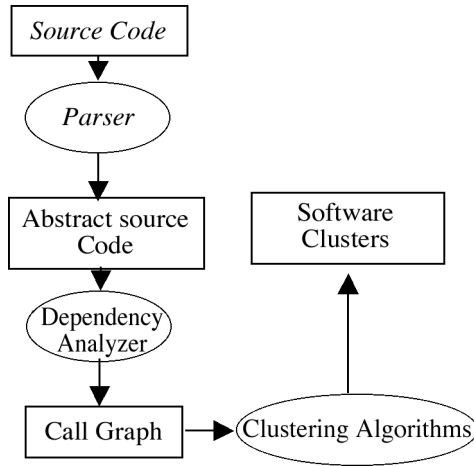


Figure 1: Software Clustering Process

**Algorithm: Genetic Clustering**

```

1. Create Current Population Using Encoding operator.
2. If pre-processing is required then number graph nodes in breath first order.
3. If local optimization is required then Perform localOptimization (currentPopulation)
4. While not StopCondition() do
  for each Chromosome in current population
  5.           decode(Chromosome)
  6.           Calculate fitness(Chromosome)
  end for
7. if gender is defined then
  sort Chromosomes in order of quality
  assign gender to each Chromosome
  end if
8. Create intermediate generation using Selection
9. Crossover (c1, c2) resulting O1 and O2
10. Mutate O1 , O2
11. If local optimization is required then
  localOptimize O1 , O2
12. Create Next Generation using Reproduce
End While
  
```

Figure 2: General structure

The above algorithm makes use of a set of components which are highlighted in bold. Defining, a function with a standard interface for each of these component types, various schemes for the components may be used, without any need to change the body of the algorithm. For instance, there are several schemes for the selection operator: roulette wheel selection, tournament, elitist models, and ranking methods. Defining a unique interface for the selection function each of these schemes may be used without any need to modify the rest of the genetic algorithm which makes use of the selection function.

**4. GENETIC COMPONENTS**

Genetic clustering algorithms mostly use a fixed set of components with the same objectives but different schemes. Hence, by defining a general algorithm, as described in Figure 2, and a standard

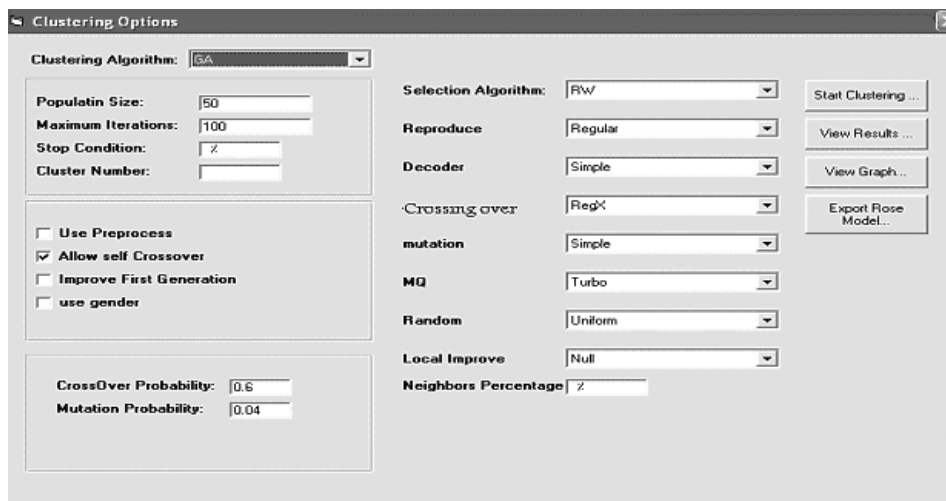


Figure 3: DAGC main user interface

interface for each component accessed by the algorithm, a flexible code can be developed for these algorithms, such that each component may be easily changed or modified without any need to modify the rest of the source code developed for the algorithm.

The DAGC user interface to access the components is illustrated in Figure 3. The main window is divided into six sections. Each section allows the user to select a predefined or new implementation of a GA component. These components are described in the following sections.

#### 4.1 Encoding Operator

The first step in genetic algorithms is translating the phenotype space to a genotype space using an appropriate representation. The maximum size and variety of the Chromosomes population is dependent on the encoding approach. Good encoding is probably the most important factor for the performance of a genetic algorithm.

In DAGC, three different schemes for encoding a clustering are presented, as default. These default schemes are Simple Encoding (a usual encoding for clustering problem) (Mitchell, 2002), ODPX (Cincotti, 2002) and the third one which has been developed by the authors and is described in Section 6.1. In addition to these three schemes, users have possibility of defining their own encoding scheme and augmenting it to the list of existing schemes by implementing the following interface:

```
public void Decode ( Chromosome C, Graph G, Int NoClusters, Vector NodeSequence)
```

In the above definition, *NodeSequence* is a Vector which defines a mapping between the gene positions on a chromosome and their corresponding graph nodes. This vector is filled by the preprocess function, described in the Section 4.3. This interface simply translate chromosome C into graph G. Parameter *NoClusters* controls maximum number of clusters in the graph(optional).

#### 4.2 Random Generation

The first population of chromosomes is generated randomly. Each individual is in fact generated by random partitioning of the genes. To achieve this, a function called randomized is defined as follows:

```
public void randomize(Chromosome c, int k);
```

We have defined two different implementations of this interface in DAGC: uniform and permutation. Uniform random generation assigns each graph node a cluster number based on a uniform probability. The permutation random generator creates a random permutation of graph nodes (Section 5.1).

### 4.3 Fitness Function

In order to estimate the fitness or the quality of a clustering, two known program modularization criteria, BasicMQ and TurboMQ (Mitchell, 2002) can be accessed as default fitness functions within the DAGC environment. The BasicMQ fitness function measures inter-connectivity (i.e., connections between the components of two distinct clusters) and intra-connectivity (i.e., connections between the classes of the same cluster) independently.

The BasicMQ objective function is designed to produce higher values as the intra-connectivity increases and the inter-connectivity decreases. This function is defined as follows:

$$\text{basicMQ} = \frac{1}{k} \sum A_i - \frac{1}{\frac{k(k-1)}{2}} \sum E_{ij} \tag{1}$$

Which  $A_i$  is the intra-connectivity and  $E_{ij}$  is the inter-connectivity:

$$A_i = \frac{\mu_i}{N_i^2} \tag{2}$$

$$E_{ij} = \frac{\epsilon_{ij}}{2N_iN_j} \tag{3}$$

in the above definitions,  $N_i$  is the number of graph nodes in the cluster  $i$ ,  $\mu_i$  is the number of intra-edge relations in cluster  $i$  and  $\epsilon_{ij}$  is the number of inter-edge relations of cluster  $i$  and  $j$ . BasicMQ function can not be used for weighted graphs (Mitchell, 2002).

Another useful fitness function is TurboMQ. This function can be used for weighted graphs (Mitchell, 2002) and is defined as follows:

$$\text{turboMQ} = \sum CF_i \tag{4}$$

Formally, the TurboMQ measurement for a partitioned graph into  $k$  clusters is calculated by summing the Cluster Factor (CF) for each cluster in the partition. CF for cluster  $i$  is defined as follows:

$$CF_i = \frac{2\mu_i}{2\mu_i + \epsilon_i} \tag{5}$$

As before,  $\mu_i$  is the number of intra-connectivity and  $\epsilon_i$  is the number of inter-connectivity for cluster  $i$ . This Objective function approximates the cohesion and coupling of each cluster independent of other clusters using  $CF$  definition. The user may define any fitness function, by implementing the following interface:

```
public float Calculate_Fitness( Graph G, Chromosome C, Vector Constraints);
```

Objective functions can not solely satisfy all the user's requirements. There may be certain constraints imposed by the user, depending on the specific problem. Simple examples of constraints that could be specified by a user are: minimum and maximum numbers of clusters, bounds on the number of nodes in each cluster, specific nodes sharing a cluster and limits on the variation in cluster size.

In DAGC each constraint may be defined as a class implementing an interface class, called constraint. This class has a method with following signature:

```
Public float compute_constraint(Graph g)
```

This method is implemented for each constraint,  $C_i$ , separately and returns a value  $d_i$ , which shows the distance of the actual clustering from the optimal solution for the constraint. The fitness for a given chromosome is calculated as follows:

$$\text{FitnessValue} = \text{Objective\_function\_returned\_value} - \sum(w_i * d_i)$$

Where  $w_i$ , represents the relative importance of the constraint  $C_i$ .

#### 4.4 Preprocess

Preprocess is a function which rearranges the position of genes on chromosomes in order of their similarity. Based on schema theorem (Goldberg, 1989; Whitley, 1995), schemas with shorter defining length have higher survival chance during multiple generations (Whitley, 1995). Obviously, by keeping the most related vertices close to each other (for example by traversing the graph in breadth first order), in a chromosome, the 'good' schemas will have shorter lengths and the probability of losing quality when applying crossover to the chromosome will be reduced (Bui and Moon, 1996). In DAGC, user can define a preprocess method based on his encoding and activate this feature by selecting a check box in Figure 3. It should be noted that preprocessing heuristic is only applicable to Locus based strings, called *locus-based* encoding. Locus-based encodings for graph problems mostly map each vertex to a fixed corresponding position (locus) on the linear string (De Jong and Spears, 1989; Moon, 1994).

#### 4.5 Reproduce

The reproduce operator is applied to the current population to create the next generation. In DAGC two different methods for Reproduce are available. A *Regular* reproduce replaces the whole population with intermediate population while the *Steady state* (Bui and Moon, 1996) reproduce only replaces one parent with one of the offsprings. In DAGC, reproduction is implemented with elitism.

#### 4.6 Selection

The selection operator is used to determine which individuals in the population are candidates for reproduction based on their fitness value. Three selection schemes, called tournament, roulette wheel (Fitness Proportional) and Linear ranking (Goldberg, 1989; Whitley, 1995) are provided, as defaults, within the DAGC. New selection schemes may also be defined and augmented to the existing ones. Different selection algorithms use the following general interface:

```
Public Chromosome Select (Population P);
```

#### 4.7 Recombination Operator

*Recombination or Crossover* operators are used to create a new offspring chromosome by combining parts of the two parent chromosomes. If gender feature is assigned to each individual (Rejeb and Albelhaja, 2000), then the crossover (mating) is permitted only between individuals of opposite gender. Crossover is performed with a probability which is set by the user.

```
public void CrossOver(
    Chromosome parent_1,
    Chromosome parent_2,
    Chromosome offspring_1,
    Chromosome offspring_2,
    int noClusters,
    DecodMethod theDecoder
    MQfunction themq,
    Graph graph,
    Vector graphNodeSeq,
    double probability);
```

#### 4.8 Mutation Operator

In DAGC we have predict two types of the mutation operators: first, a *simple mutation*, in which a random graph node jumps into a random cluster and a *swap* mutation in which two random genes swap their values. User can define any new mutation by implementing the following interface:

```
public void Domutation(
    Chromosome c,
    DecodMethod theDecoder,
    MQfunction themq,
    int NoClusters,
    Graph graph,
    Vector graphNodeSeq,
    double probability);
```

#### 4.9 Local Improvement

Local improvement procedures such as KL (Bui and Moon, 1996) and FM (Bui and Moon, 1996) may be incorporated into genetic clustering algorithms to improve their performance. We used a neighborhood based method for local improvement. In Figure 3, a combo box labeled *local improve* is assigned to select an existing function or create a new local improvement function. The user can define a new Local Improvement method by implementing the following general interface:

```
public Chromosome LocalImprovement(
    Chromosome c,
    DecodMethod theDecoder,
    MQfunction themq,
    int maxNeighbors,
    Vector graphNodeSeq,
    int NoClusters,
    Graph g);
```

#### 4.10 Stop Condition

As illustrated in Figure 3, there are two input texts for defining stopping conditions in DAGC. One input is the maximum number of generations and the other one defines the percentage of similarity of individuals in a generation as a termination criterion for the corresponding genetic algorithm.

### 5. ARCHITECTURE

As shown in Figure 4, the DAGC architecture consists of three interacting layers: API, Driver and Services which are described below.

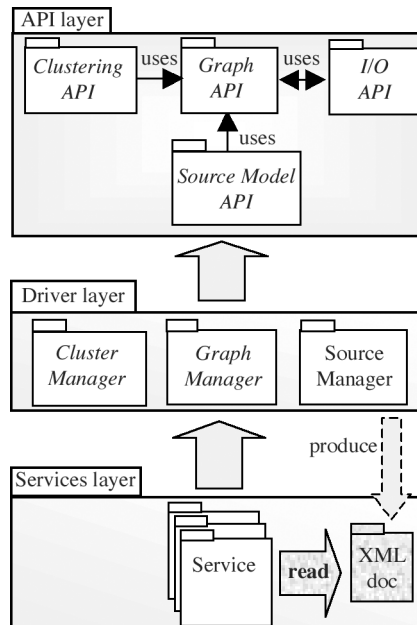


Figure 4: The Architecture of the DAGC Tool

### 5.1 API Layer

The API layer is an independent layer which includes a useful and complete set of java classes. This library has most crucial classes for design recovery applications. These classes are arranged in four packages as follows:

- ClusteringAPI: The ClusteringAPI package provides useful classes and interfaces for the genetic components described in Section 4.
- GraphAPI: The GraphAPI package provides a set of classes which can be used to generate and display three kinds of benchmark graphs called Random, Caterpillar and software graphs (Bui and Moon, 1996). These three types of graphs are used for evaluation and assessment of the genetic algorithms developed within the DAGC environment. Random Graphs are shown by  $R.n.d$  symbol. Here,  $n$  is the number of graph nodes and  $d$  is the expected degree of each node in that graph (for example  $R.30.2$  defines a graph with 30 nodes and expected degree 2 for each node). A simple algorithm can be used for creating a Random graph: First we produce  $n$  different nodes and then place an edge between each two nodes with a probability  $P$ . A proper value for  $P$  in graph  $R.n.d$  may be  $d/(n-1)$ . Software graphs are special types of Random Graphs. Experiments with several software graphs show that in these graphs the number of edges have a linear relation with number of nodes,  $(O(|E|)=O(|V|))$ . A software graph  $S.n.k$  has  $n$  nodes and  $n*k$  edges.
- I/O API: This package provides a number of useful classes for loading and writing graph files. Here, we support three formats for importing or exporting graph files which are: Dotty-AT&T, XML and Text (Mitchell, 2002).
- SourceModelAPI: This package contains a number of classes to extract call graphs from a Java source code. To extract class dependency graphs three different methods called CHA, RTA (Sundaresan and Hendren, 2000) and FRTA(Flow sensitive Rapid Type Analysis) can be

selected in DAGC. These algorithms present different heuristics to resolve polymorphic calls. RTA and CHA are well-known methods for resolving polymorphic calls. FRTA is the DAGC algorithm for extracting call graphs based on RTA. In fact, FRTA is a modified RTA which traverses program and creates *instantiated-types* (Sundaresan and Hendren, 2000) incrementally based on probabilistic program flow.

### 5.2 Service Layer

The service layer provides all the interfaces of the DAGC framework. At this moment DAGC presents the following services:

- *Clustering Service*: This service allows the user to create a new GA. This GA is submitted to the Driver Layer using an associative array to map parameters to values to build and execute.
- *Data Analyzer Service*: This service provides facilities to compare and assess the clustering results based on execution time and final quality. Furthermore this service employs the *confidence analysis* method (Section 6) to build a consolidated model.
- *Customizing Service*: This service provides some templates for genetic components and allows us to customize them for building our new component. DAGC uses Java Dynamic Class Loading feature to load our customized component in GA.

### 5.3 Driver Layer

The driver layer assembles appropriate GA components according to the parameters set by the user. The clustering results are stored in a XML document, by this layer. The XML document is used by the Service layer or any other data analyzer to perform arbitrary analysis on the clustering results.

## 6. DAGC CLUSTERING ALGORITHM

In this section a new clustering algorithm is presented. The algorithm is based on our new encoding and recombination schemes. To develop the algorithm we started with an algorithm called Bunch (Mitchell, 2002). In the DAGC environment we could easily replace the Bunch encoding scheme with our new encoding scheme, and immediately observe the results. To improve the algorithm, different schemes for the components, described in Section 4, were tried and the results were evaluated. Some of the results are presented in the following subsections.

### 6.1 Permutation-based Encoding

In our new encoding scheme, each chromosome is a permutation of  $N$  integers. Here, the  $m^{\text{th}}$  gene of the chromosome, instead of holding a partition number  $k$  (which means node  $m$  of the graph resides in partition  $k$ ), holds a value  $1 \leq p \leq N$ . A chromosome can be decoded into a clustering by the following algorithm:

```
Algorithm Decode chromosome
  Array C holds a permutation of graph nodes 1 to N
  For node i = 1 to N do
    If C[i] >= i Create a new cluster and assign node i to it.
    Else
      Allocate i to the same cluster as node C[i].
  End Decode
```

Hence, considering a graph with  $N$  vertices, any permutation of the numbers 1 to  $N$  can represent a clustering for the graph. As an example consider the permutation:

$P = 1\ 5\ 6\ 3\ 2\ 4$

Using our encoding scheme, vertex 1 will belong to partition 1. The second value in P is 5 which is greater than 2, hence vertex 2 is assigned to partition 2. Vertex 3 is assigned to partition 3. Vertices 4, 5 and 6 are assigned to the same partitions as vertices 3, 2 and 4 respectively, because the fourth, fifth and sixth values in P are 3, 2 and 4. The result of the clustering is shown in Figure 6.

To encode a clustering into a chromosome the algorithm presented in figure 5 is used.

```

Algorithm Encode
Input: A Graph with N nodes and K clusters
Output: A chromosome C with N genes
Begin
  Create an empty chromosome C with N genes.
  For i=1 to k
    A is an array containing nodes in cluster #i
    p is the number of nodes in A
    Sort A, in ascending order
    For j=1 to p-1
      Assign A(j) to C(A(j+1))
    Next j
    Assign A(p) to C(A(1))
  Next i
End
    
```

Figure 5- DAGC encoding algorithm

In Figure 6 a chromosome and its corresponding clustered graph are shown.

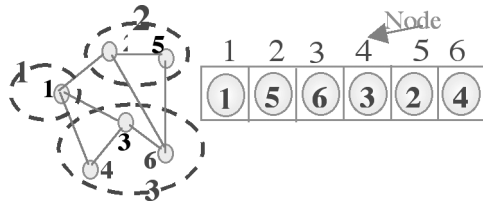


Figure 6: A partition and its corresponding string

By inspecting the above algorithm, it is clear that for a given graph with n nodes, the search space for GA may include up to n! strings which is much smaller in comparison with the n<sup>n</sup> strings produced by the simple encoding (Mitchell, 2002) scheme. This makes solving the problem easier for the GA. Reduction in the search space helps the GA to converge faster and to find acceptable solutions during earlier generations. Figure 7 shows a comparison between our new algorithm (DAGC algorithm) and the Bunch algorithm. The former uses our permutation encoding and the latter uses a simple encoding. In this experiment, the input graph was a software graph with 100 nodes. For each generation we had 10 runs and the fitness values in Figure 7 are average of these 10 outcomes calculated by TurboMQ fitness function.

As shown in Figure 7, our clustering algorithm converges faster when using our permutation based encoding scheme, compared with when simple encoding is used. For example, the simple encoding scheme yields fitness of 2.0 after 120 generations while our permutation encoding reaches to this fitness after 80 generations.

Since our coding is a permutation, we need an order based recombination operator to preserve

the permutation form in each offspring. Here, a one point recombination operator is used.

Suppose there are two parents P1 and P2 as follows:

P1:  $C_1C_2 \dots C_kC_{k+1}C_{k+2} \dots C_L$

P2:  $D_1D_2 \dots D_kD_{k+1}D_{k+2} \dots D_L$

To create two offspring O1 and O2, first, parents are split at position k. First parts of two offspring are created using the first parts of two parents:

O1:  $C_1C_2 \dots C_k$

O2:  $D_1D_2 \dots D_k$

To fill the second parts of two offspring, we create the following list:

$C_{k+1} D_{k+1}C_{k+2} D_{k+2} \dots C_L D_L$

Starting from the first element in the list, if the element does not belong to the first offspring, O1, it will be augmented to this offspring. Otherwise, it will be added to the second offspring. Analogously, we can consider the list

$D_{k+1} C_{k+1}D_{k+2} C_{k+2} \dots D_L C_L$

and generate two other offspring.

### 6.2 Performance of DAGC Algorithm

To evaluate our new encoding scheme, the Bunch algorithm was assembled, using the clustering interface, illustrated in Figure 3. Then a new algorithm, called DAGC, was created by simply replacing the encoding and recombination methods of the Bunch algorithm with ours. Since the DAGC GA works on a much smaller search space, better clustering quality were expected when using TurboMQ or BasicMQ fitness functions.

#### 6.2.1 Experiments with TurboMQ Fitness function

To compare our new genetic algorithm, DAGC, with Bunch, nine different graphs with more than 200 vertices were used. Each graph was clustered five times, using both the DAGC and Bunch clustering algorithms. The average and variance for the results are shown in Figures 7 and 8 respectively. By definition (Mitchell, 2002), a clustering algorithm is unstable if repeated runs of the algorithm on the same graph produce results that vary significantly. Figure 7 demonstrates the

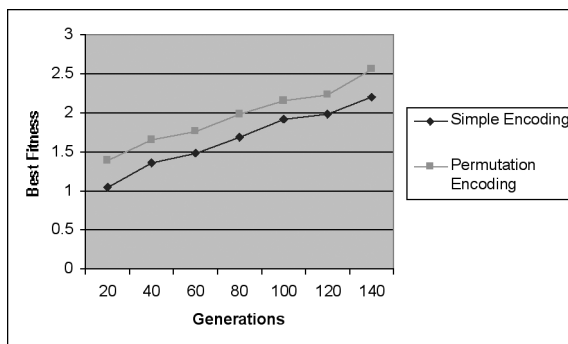


Figure 7: A comparison between codings: The permutation algorithm produces better solutions during generations.

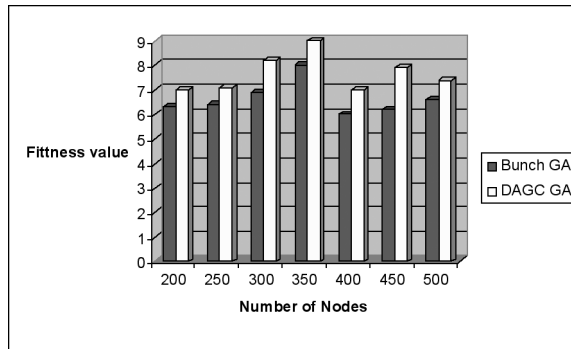


Figure 8: Comparing quality of results with TurboMQ fitness function

stability of the DAGC encoding scheme in comparison with the encoding scheme used by the Bunch.

To evaluate the performance of our permutation-based coding scheme the TurboMQ fitness function was used with the Bunch algorithm. The algorithm was then executed twice. Once with its original coding scheme and once with our coding scheme. A comparison of the clustering results is presented in Figure 8. The y-axis in this figure shows the objective function value for the clustered graph. As shown in Figure 8, due to the relatively high reduction in the search space that results from applying our permutation-based coding scheme, clustering with higher quality was produced.

### 6.2.2 Experiments with BasicMQ Fitness function

The BasicMQ fitness function calculates the quality of a class dependency graph partitioning by subtracting the average cohesion and average coupling of the clusters. Since, the number of clusters does not affect the evaluation of BasicMQ fitness function, better results were produced, when using the BasicMQ fitness function instead of the TurboMQ. Figure 9 shows the results of executing DAGC and Bunch GAs on five random graphs, using the BasicMQ fitness function.

As depicted in Figure 9, the quality of clustering reduces when increasing the number of graph nodes. Our experiments with DAGC and Bunch clustering algorithms show that for a given graph, DAGC clusters the graph much faster. Figure 10 shows the execution time for applying these two algorithms on different graphs. For executing Bunch GA we have used Bunch API that is available

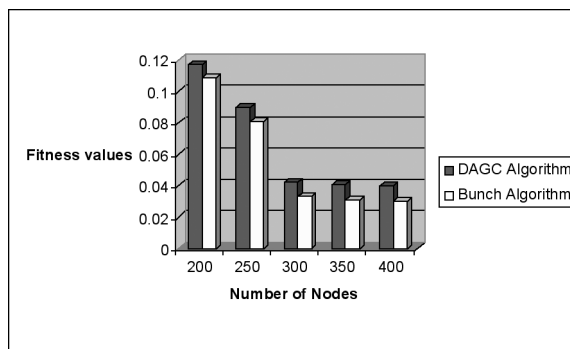


Figure 9: Comparing quality of results with BasicMQ fitness function

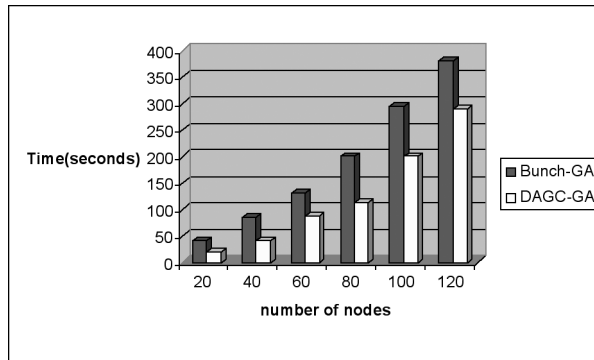


Figure 10: Performance of the Bunch and DAGC algorithms

online.

## 7. COMMON MODELS

Due to the heuristic nature of genetic algorithms, they may produce different results in different runs. Genetic clustering algorithms may produce results with the same quality but different partitioning for different runs on a given graph. Here, the question is on which criteria a partitioning should be selected for a given graph.

In Mitchell and Mancoridis (2001b) an approach called *Confidence analysis* is proposed to produce a consolidated model from different clustering results produced by several runs of an algorithm or different algorithms on a given graph. The common model may be used as a reference decomposition to evaluate each clustering result.

To perform Confidence Analysis, first a given graph is clustered many times. Each clustering is represented as a set of relations  $(n_i, n_j)$ , where  $n_i$  and  $n_j$  are two nodes in the same cluster. Then, a consolidated model using all the clustering results is created. This model is a set of triples  $(n_i, n_j, c)$ , where  $c$  is the percentage of the number of times that nodes  $n_i$  and  $n_j$  have been observed in the same cluster. If this value is greater than a given threshold, then the two nodes will be assigned to the same cluster in the common model.

In DAGC a call graph can be clustered using different algorithms. The result of each clustering may be saved in a database table. DAGC creates a common model by applying confidence analysis (Mitchell and Mancoridis, 2001b; Mitchell, 2002) to the clustering results saved in the table. The common model may be used as a reference decomposition. Also, DAGC can accept any clustering as a reference decomposition to compare with the common model or any clustering result saved in its database.

We applied different clustering algorithms to dependency graphs created for three software systems: *Compost* (<http://www.the-compost-system.org>, 2003) library with 32 JAVA packages and up to 500 classes, *Apache-ant* (<http://ant.apache.org>, 2004) which contains a set of java packages for web server programming and finally a library called *Swing* (<http://javasoft>, 2002). For these three software libraries the original modularizations are available.

To see the effect of a threshold value on the final clustering modularity, we have produced the common model for our case studies using different confidence values. Comparing each of these partial clustering with their original decomposition, shows that higher confidence values produce relatively higher modular partitions. The comparison is based on the Precision/Recall (Mitchell and

<b>Result#</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>	<b>11</b>	<b>12</b>	<b>13</b>	<b>14</b>
<b>P/R %</b>	19/9	30/14	31/16	33/41	32/47	33/35	32/18	24/11	33/17	22/10	32/15	27/12	27/10	30/20
<b>Quality</b>	15.7	15.6	14.8	16.6	16.0	16.0	16.1	15.8	16.0	15.4	15.3	15.1	15.3	15.4
<b>Result#</b>	<b>15</b>	<b>16</b>	<b>17</b>	<b>18</b>	<b>19</b>	<b>20</b>	<b>21</b>	<b>22</b>	<b>23</b>	<b>24</b>	<b>25</b>			
<b>P/R %</b>	24/11	31/17	25/9	32/15	24/11	32/18	22/9	32/15	27/12	26/11	29/20			
<b>Quality</b>	16.0	16.0	15.7	15.3	16.0	16.1	15.7	15.3	15.0	15.3	15.5			

Figure 11: Evaluating clustering results(P/R=Precision/Recall)

Mancoridis, 2001a) criteria. In the other words, increasing the confidence value omits the *wrong cluster-mate* relationships from our model.

In Figure 11, the results of analysing the *Compost* class library are shown. The *Compost* was clustered 25 times. For each run, the quality of the clustering based on TurboMQ, and the similarity between the clustering and the original decomposition of the *Compost* is presented. These results were used to create a common model.

The common model was created, using nine different thresholds. The results of similarity measurements between the common model and the original decomposition for the *Compost* are depicted in Figure 12.

In the first row of the table presented in Figure 13, the highest similarity value for different runs of applying clustering algorithms to the three software systems, with their original decompositions is presented. The second row shows the similarity between the common model produced for each set of the clustering results and the original decompositions for confidence value 60%. As it was expected, the common models are more similar to the original decompositions than individual clustering results.

In fact choosing cluster-mates which have a higher frequency in the consolidate model, increases the inter-connectivity of the final model.

### 8. CONCLUSION

It is possible to develop frameworks to assemble genetic clustering algorithms by defining standard

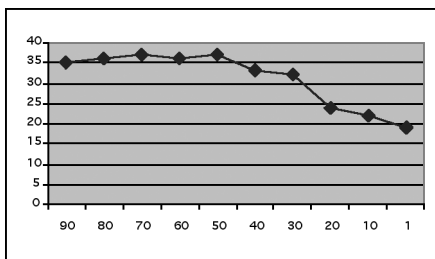


Figure 12: Variation of Precision for different thresholds for Compost

<b>Threshold</b>	<b>90</b>	<b>80</b>	<b>70</b>	<b>60</b>	<b>50</b>	<b>40</b>	<b>30</b>	<b>20</b>	<b>10</b>	<b>1</b>
<b>Precision</b>	35	36	37	36	37	33	32	24	22	19
<b>Recall</b>	1	2	3	3	3	4	5	25	48	56

	<b>Compost</b>	<b>Apache</b>	<b>Swing</b>
Different results	33%	35%	20%
Common model	36%	37%	29%

Figure 13: Comparison results

interfaces for the building blocks. DAGC is an example of a flexible environment to survey clustering algorithms for software re-modularization. DAGC provides a flexible library of genetic components and algorithms. The users can assemble new algorithm or modify existing ones by simply selecting the components from the DAGC component library and observe the results of changing the components. New schemes for the components can be augmented to the DAGC component library, by the users.

The DAGC framework can create reference decompositions for evaluating the results of genetic clustering algorithms. The framework can also accept an existing decomposition as a reference for evaluating the results. Using this feature of the DAGC environment, it was demonstrated that DAGC could offer better modular architectures than the existing ones for the three well known software packages, Swing, Compost and *Apache-ant*.

The experimental results of this paper, show that our permutation based encoding scheme produces clustering with higher fitness values in comparison with other encoding schemes conventionally used for genetic clustering.

### REFERENCES

- ABREU, F.B., PEREIRA, G. and SOUSA, P. (2000): A coupling guided cluster analysis approach to reengineer the modularity of object oriented systems, *Conference on Software Maintenance and Reengineering*, IEEE.
- ANQUETIL, N. and LETHBRIDGE, T. (1999): Experiments with clustering as a software remodularization method, *The Sixth Working Conference on Reverse Engineering (WCRE'99)*.
- BUI, T.N. and MOON, B.R. (1996): Genetic algorithm and graph partitioning, *IEEE Trans. Comput.*, 45(7): 841–855.
- CINCOTTI, A., CUTTELO, V. and PAVONE, M. (2002): Graph partitioning using genetic algorithms with ODPX, *Proceedings of the World Congress on Computational Intelligence*, IEEE.
- DAVEY, J. and BURD, E. (2000): Evaluating the suitability of data clustering for software remodularisation, in the *Proceedings of Seventh Working conference on Reverse Engineering*, IEEE.
- DEJONG, K. and SPEARS, W. (1989): Using genetic algorithms to solve NPcomplete problems, in *3rd Int. Conf. on Genetic Algorithms*, 124–132.
- GAREY, M.R. and JOHNSON, D.S. (1979): Computers and intractability: A guide to the theory of NP-Completeness, Freeman.
- GOKEL, N., DRECHSLER, R. and BECKER, B. (1997): GAME: A software environment for using genetic algorithms in circuit design, *Applications of Computer Systems*, 240–247.
- GOLDBERG, D. (1989): Genetic algorithms in search, optimization and machine learning, Addison Wesley.
- HEITKOTTER, J. and BEASLEY, D. (2000): The hitch-hiker's guide to evolutionary computation (FAQ for cmp.ai.genetic). <<http://www.cs.bham.ac.uk/Mirrors/ftp.de.uu.net/EC/dife/www/>> March.
- <<http://www.the-compost-system.org>> (2003)
- <<http://www.javasoft.com>> (2002)
- KEIJZER, M., MERELO, J. and ROMERO, G. (2002): Evolutionary objects: a general purpose evolutionary computation library, University of Granada, Spain.
- MANCORIDIS, S., MITCHELL, B. S. and RORRES, C. (1998): Using automatic clustering to produce high-level system organization of source code, in the *Proceedings of the International Workshop on Program Understanding (IWPC'98)* Italy, IEEE.
- MITCHELL, B.S. (2002): A heuristic search approach to solving the software clustering problem, *Thesis*, Drexel University, March.
- MITCHELL, B.S. and MANCORIDIS, S. (1999): Bunch: A clustering tool for the recovery and maintenance of software system structure, in *Proc. of International Conf. of Software Maintenance*, IEEE.
- MITCHELL, B.S. and MANCORIDIS, S. (2001a): Comparing the decompositions produced by software clustering algorithms using similarity measurements, in the *Proceedings of international conference on software maintenance (ICSM'01)*, Italy, IEEE.
- MITCHELL, B.S. and MANCORIDIS, S. (2001b): CRAFT: A framework for evaluating software clustering results in the absence of benchmark decomposition, in *Proc. of IWPC*, IEEE.
- MOON, B.R. (1994): Hybrid genetic algorithms with hyperplane synthesis: A theoretical and empirical study, *Ph.D. dissertation*, Pennsylvania State Univ., University Park.
- REJEB, J. and ALBELHAJIA, M. (2000): New gender genetic algorithm for solving graph partitioning problems, *Proc. 43rd IEEE Midwest Symp. On Circuits and Systems*, Lansing MI, August 9–11.
- SAEED, M., MAQBOOL, O., BABRI, H.A., HASSAN, S.Z. and SARWAR, S.M. (2003): Software clustering techniques and the use of combined algorithm, *Proceedings of the Seventh European Conference On Software Maintenance and*

*Reengineering* (CSMR'03) IEEE.

SUNDARESAN, V. and HENDREN, L. (2000): Practical virtual method call for java, Proceedings of the conference on Object-Oriented programming, systems, languages, and applications.

WALL, M. (1995): Overview of GALIB, <http://lancet.mit.edu/ga>

WHITLEY, D. (1995): A genetic algorithm tutorial, Computer Science Department, Colorado State University.

WIGGERTS, T.A. (1997): Using clustering algorithms in legacy systems remodularization, *Fourth Working Conference on Reverse Engineering*.

WOODRUFF, D.L. (2002): Optimization software class libraries, Handbook, March.

### BIOGRAPHICAL NOTES

*Saeed Parsa received his BSc in mathematics and computer science from Sharif University of Technology, Iran, his MSc degree in computer science from the University of Salford in England, and his PhD in computer science from the University of Salford, England. He is an associate professor of computer science at Iran University of Science and Technology. His research interests include software engineering, soft computing and algorithms.*



Saeed Parsa

*Omid Bushehrian received his BSc in software engineering from AmirKabir University of Technology (Tehran Polytechnics), Iran, his MSc degree in software engineering from University of Science and Technology, Iran. He is now a PhD student in software engineering at Iran University of Science and Technology. His research interests include distributed systems and clustering algorithms.*



Omid Bushehrian