Genetic Clustering with Constraints

Saeed Parsa and Omid Bushehrian
Department of Computer Engineering,
Iran University of Science and Technology
{parsa, bushehrian}@iust.ac.ir

The aim is to facilitate the application of user defined constraints to the genetic clustering algorithm. This is achieved by presenting a general penalty function. The penalty function is defined as a normal distribution. The function is augmented to an extensible environment to assemble genetic clustering algorithms, called DAGC. The main idea behind the design of DAGC is to provide the researches with an environment to develop and investigate genetic clustering algorithms by selecting the building blocks from an extensible library. It also provides the user with some templates to build their own building blocks. This new version of DAGC is equipped with some interfaces to define new constraints or to apply existing ones.

ACM Classification: G.3 (Mathematics of Computing, Probability and Statistics, Probabilistic algorithms)

1. INTRODUCTION

Graph clustering plays an important role in automatic distribution of sequential code (Kimelman *et al*, 1997; Tilevich and Smaragdakis, 2002) and program re-modularization (Mitchell and Mancoridis, 1999; Mitchell, 2002; Parsa and Bushehrian, 2004; Parsa and Bushehrian, 2005a; Parsa and Bushehrian, 2005b). We have developed a tool, called DAGC (Parsa and Bushehrian, 2004; Parsa and Bushehrian, 2005a; Parsa and Bushehrian, 2005b) to extract dependency graphs from Java source code and cluster the graphs. The tool follows a framework where genetic clustering algorithms can be assembled by simply selecting the building blocks from an extensible library. In this paper a new extension of the DAGC to impose user defined constraints on graph clustering algorithms is presented.

Graph clustering efforts have mostly focused on clustering and decomposition techniques based on high intra-cluster or cohesion and low inter-cluster or coupling (Anquetil and Lethbridge, 1999; Mitchell and Mancoridis, 1999; Mitchell, 2002; Parsa and Bushehrian, 2004; Parsa and Bushehrian, 2005b). However, there are situations where users want to impose their own constraints on the clusters. The most usual approach to incorporate constraints into the genetic clustering process is the penalty functions (Crossley and Williams, 1997; Coello, 2002). The method by which infeasible individuals are penalized has a significant effect on their final fitness values and should be defined carefully. In this paper an adaptive approach to apply constraints is proposed.

Within the DAGC environment users may define multiple constraints. This has been achieved by augmenting the DAGC with a new component with a standard interface to define constraints. In DAGC, there are three implementations of the component as defaults to impose the following constraints:

Manuscript received: 13 August 2005
Communicating Editor: John Yearwood

1. Clusters to be balanced
2. A number of nodes share the same cluster
3. Limit the number of clusters to a given range

In addition to the above mentioned constraints, users may define their own constraints and insert them into the DAGC component library. To define a new constraint, an interface and a template is presented in Section 3.1.

The remaining parts of this paper are organized as follows: Section 2, provides a detailed description of our proposed normal distribution penalty function. The standard DAGC interface and templates for defining constraints is presented in Section 3. Three examples of defining constraints by implementing the proposed interface are presented in Section 4. Examples of applying the constraints are also presented in Section 4. In Section 5 a general constrained genetic clustering algorithm is presented. The algorithm invokes genetic parts with standard interfaces. This algorithm is implemented in the DAGC environment, as a flexible tool to investigate constrained genetic clustering algorithms.

2. RELATED WORKS

Constraints are essential parts of the clustering process. The most popular approach to apply constraints is to define penalty functions (Gen and Chang, 1997; Crossley and Williams, 1997; Coello, 2002; Homaiffar *et al*, 1994; Joines and Houck, 1994). The penalty function could be assumed as a distribution of the distances of infeasible individuals in a problem search space from the feasible region. In general, there are three classes of penalty functions called quadratic, linear and step linear (Crossley and Williams, 1997). These functions could be applied adaptively by the means of a draw-down coefficient. The draw-down coefficient could be either a constant (Homaiffar *et al*, 1994) or a variable whose value is increased as the generation number grows (Crossley and Williams, 1997; Joines and Houck, 1994). This approach does not adapt the penalty value to the relative characteristics of successive generations. To resolve the difficulty, the variance of the population fitness in each generation is considered as the draw-down coefficient (Crossley and Williams, 1997). In this strategy as the variety of the individuals fitness values grows the variance increases and more penalties are applied to the individuals. However in this strategy, the variance of a homogeneous population will be the same whether the individuals are feasible or infeasible whilst the amount of the penalty should vary depending on the degree of feasibility of the individuals. Hence, it is observed that the feasibility of a population affects the amount of the penalty. In the strategy proposed in this paper, the draw-down coefficient is defined as a function of the infeasibility of the genetic population where the infeasibility of each chromosome is defined as the distance of that chromosome from the feasible region. In the next section the overall description of our proposed penalty function is presented.

2.1 Overall Description

The penalty value for each solution within a genetic population of solutions can be considered as a function of the distance of the solution from the feasible area where all the solutions satisfy the constraint (Crossley and Williams, 1997). Obviously the distance of a feasible solution from the feasible area is zero. If we consider a normal distribution, $N(d(x), \mu, \sigma)$ for the penalty values, then the mean value of the normal distribution should be zero because all the distances are computed relative to the mean value, μ, which is the value for the solutions in the feasible area. The mean of the normal distribution, μ, represents a feasible solution because the value of the normal

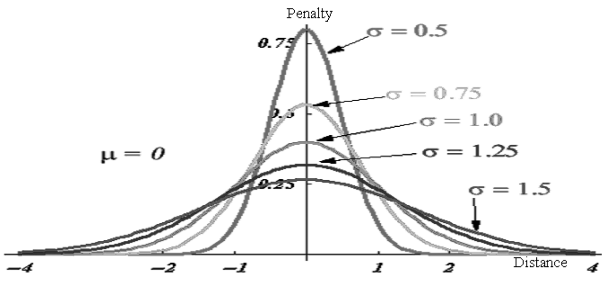


Figure 1: The effect of varying σ_t in a normal distribution

distribution function at μ, N(μ, μ, σ), is maximum on the other hand the penalty value for a feasible solution in our clustering algorithm, which is a maximization problem, as shown in relation (6) is the maximum value of the penalty function.

The standard deviation, σ, is a measure of the feasibility of the genetic population, because it is equal to the sum of the distances of solutions within the genetic population because it is the mean deviation of distances of individual solutions within the genetic population from the feasible region. As the number of feasible solutions in a genetic generation increases the value of σ reduces and as shown in Figure 1, the skewness of the normal distribution increases and as a result more penalty values will be calculated for the less feasible solutions. In this way, recalculating the value of σ for each generation the penalty function adopts itself to the ratio of infeasible solutions within the population.

2.2 Penalty Function

Considering a given constraint, the population of chromosomes in each generation can be divided into two groups of feasible and infeasible solutions. An infeasible solution does not satisfy the desired constraint. The degree of infeasibility of a solution, c, is considered as the distance, d(c), of its constraint value, g(c), from the feasible region. Here, we have considered a normal distribution function for computing a penalty value from a given distance value. For instance, if the constraint is to balance the number of nodes residing in each cluster of a clustering, c, then g(c) can be defined as the mean deviation of the number of nodes in each cluster of c. Obviously for a feasible solution $c_{feasible}$, $g(c_{feasible})=0$. The un-scaled penalty value u_p(c) for each solution, c, is computed as follows:

$$u\_p(c) = N(d(c), 0, \sigma_t) \quad (1)$$

$$\sigma_t^2 = \sum d(c_i) \ , \forall c_i \in \text{ population in genration } t \quad (2)$$
$$d(c_i) = g(c_i) - g(c_{feasible})$$

If the quadratic and linear (Crossley and Williams, 1997; Lopez-Vallejo *et al*, 2000) distributions are used, the penalty values will be calculated as follows:

$$u\_p(c) = d(c) \qquad \text{for linear distribution.} \quad (3)$$

$$u\_p(c) = d(c)^2 \qquad \text{for quadratic distribution.} \quad (4)$$

Infeasible solutions are either filtered out or penalized. To penalize an infeasible solution, c, its fitness can be reduced by a penalty factor P ∈ [0, 1] as follows:

$$P(c) = (u\_p(c) - u\_p(c_0)) \;/\; (u\_p(c_{feasible}) - u\_p(c_0)) \tag{5}$$

$$penalizedFitness(c) = estimatedFitness(c) * P(c) \tag{6}$$

In the above relation, c_0 is the solution for which the distance $d(c_0)$ from the feasible solution is maximum and $c_{feasible}$ is the solution for which the distance $d(c_{feasible})$ from the feasible solution is zero(i.e $c_{feasible}$ is a feasible solution). Using the above relations, a penalty value of 0 is assigned to the most infeasible solution and a penalty value of 1 is assigned to each feasible solution.

2.3 Adaptive Penalty

In some situations it gets very difficult to find feasible solutions which satisfy a given set of constraints. The difficulty of satisfying the constraints can be characterized by the size of the feasible region compared to the size of the genetic population. A simple approach to alleviate the difficulty is to adapt the penalty value to the proportion of the feasible region.

Obviously when you have more choices you can be stricter. Similarly, when the ratio of the feasible solutions in a generation gets higher, more penalties can be applied to less feasible chromosomes. This can be achieved by increasing the skewness or slope of the penalty function because the curvature of this function has a direct impact on the amount of the penalty to be applied on infeasible chromosomes.

For instance, in Figure 2, two normal distribution functions N(x,0,1) and N(x,0,5) are used to define the distribution of the distances. It is observed that increasing the skewness of the normal distribution from 1 to 8 the amount of u_p(x) increases for those solutions x for which d(x) is close to the origin. Hence, when the number of feasible solution in a population, t, increases the amount of σ_t in the normal distribution is decreased. In this way those solutions which are near the feasible region are less affected by the value of the penalty, P(c), and more penalty value is imposed on the more infeasible solutions. This is achieved by defining σ_t as follows:

$$\sigma_t^2 = \sum d(c_i) \quad , \forall\ ci \in \text{ population in generation t} \tag{7}$$

Therefore, recalculating σ_t for each generation t, the penalty function adapts itself to the proportion of the feasible solutions within the genetic population. In the following subsection multiple constraints are described.

2.4 Multiple Constraints

In DAGC environment to apply multi-constrains, users can define a weight level for each constraint. By default four different weight levels of low, middle, high and very high are provided by the

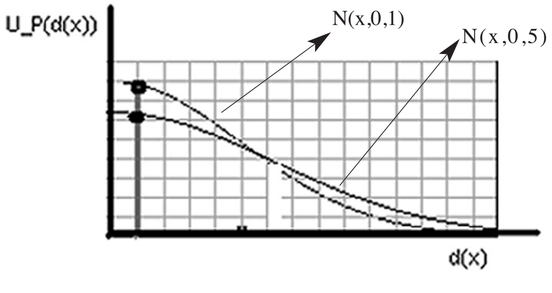


Figure 2: The effect of varying σ_t in a normal distribution

environment. In addition, users can also introduce their own weight levels in the environment. Each weight level is associated with a weight value by which the relative importance of the corresponding constraint is declared. These weight values affect the calculated penalty values for each constraint. For instance in the following example four different weight levels: low, medium, high and VHigh and their corresponding weight values 1, 2, 4 and 8 are presented using the following XML format:

```
<level1  Level="low">1</Level1>
<level2  Level="medium">2</Level2>
<level3  Level="high">4</Level3>
<level4  Level="vhigh">8</Level4>
```

Suppose that, for a given solution, C, the amount of penalties evaluated for three different constraints C1, C2 and C3 are P_1, P_2 and P_3 and they have weight levels 1, 2 and 4 respectively. The penalized fitness, F_C, for the solution will be computed using the following relation:

$$F_C = \text{Fitness}_C * (1*P_1 + \ {}^1/_2*P_2 + \ {}^1/_4 *P_3)/ (1+ {}^1/_2 + {}^1/_4)$$

In the above relation, penalty values P_1, P_2 and P_3, are multiplied by the inverse of the weight assigned to their corresponding constraints. Obviously, the higher the weight assigned to a constraint the more the constraint affects the feasibility of the chromosome and hence each penalty value is divided by the value of the weight assigned with the constraint. In the following subsection a general penalty function is presented.

3. CONSTRAINT COMPONENTS

In this section, firstly, a general template for defining constraints, using the proposed penalty function is presented. Then, the DAGC interfaces for defining the constraints are described.

3.1 Constraint and Penalty Functions

Within the DAGC environment new constraints can be defined and inserted into the DAGC component library. Each user-defined constraint adheres to the following standard interface.

```
Interface Constraint   {
          Double g (Clustering  C);    // returns the constraint value for C
          Double d (Clustering  C);    // returns the distance of solution C from the
                                       // feasible region.
          Double d0 (Clustering  C); // returns the maximum distance from the feasible
                                     // region
          Double Sigma (population thepop);  //returns the value of parameter σ
                                             //considering the no. feasible solutions
   }
```

In the above interface definition the method g(c) determines the constraint value for a given clustering c; d(c) returns the value "g(c) – g($c_{feasible}$)" for a given constraint which is the distance of the solution c from the feasible region; d0(c) determines the maximum possible distance from the feasible region; Sigma(thepop) depending on the constraint uses certain characteristics of the genetic population to work out the mean deviation of the normal distribution. Three implementations of this interface for three different constraints are presented in the following subsections. The penalty value for a given constraint is calculated with the following function:

```
Double   P(Constraint constraint_impl, Clustering  c,Population   thepop)
    { $\sigma_t$= constraint_impl.Sigma(thepop)     // get the normal distribution
                                                 // mean-deviation considering the
                                                 // population characteristics
    $d_0$= constraint_impl.d0(c)                    // getting the maximum distance
                                                 // region from the feasible
    d= constraint_impl.d(c)                      // calculating the penalty value
    u_p=N(d , 0, $\sigma_t$)
    u_pfeasible = N(0 , 0, $\sigma_t$)
    $u\_p_0$= N(d0 , 0, $\sigma_t$)
    p= (u_p - $u\_p_0$ )  / ($u\_p_{feasible}$ – $u\_p_0$ )    // scale the penalty value into range [0 1]
    return p;
}
```

The function P takes a reference to the function g(c) to evaluate the constraint value for a given solution c. P then calculates the penalty value for c using relation (5), described above. The rate of feasibility of the population affects the resultant penalty for the solution c.

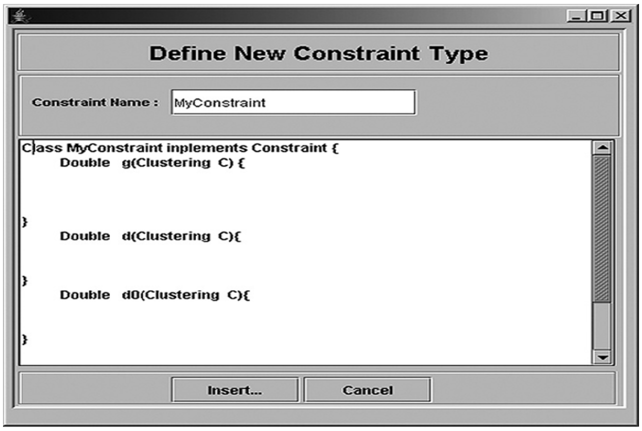
3.2 User Interfaces

Within the DAGC environment, new constraints can be defined as a class implementing the constraint interface described above in Section 3.1. For each user defined constraint class, the *Constraint* interface is implemented using the dialog box shown in Figure 3.

After a constraint is defined, it is automatically augmented to the DAGC component library. To apply the constraint to genetic clustering algorithms, created within the DAGC environment, the dialog shown in Figure 4 can be used.

4. DEFAULT CONSTRAINTS

Constraints are naturally available in many clustering applications. The problem of clustering with constraints is receiving increasing attention. Simple examples of clustering constraints are as follows:


Define New Constraint Type

Constraint Name : MyConstraint

```
Class MyConstraint inplements Constraint {
      Double  g(Clustering  C) {


}
      Double  d(Clustering  C){


}
      Double  d0(Clustering  C){


}
```

Figure 3: Defining a new user-defined constraint by implementing the *Constraint* interface

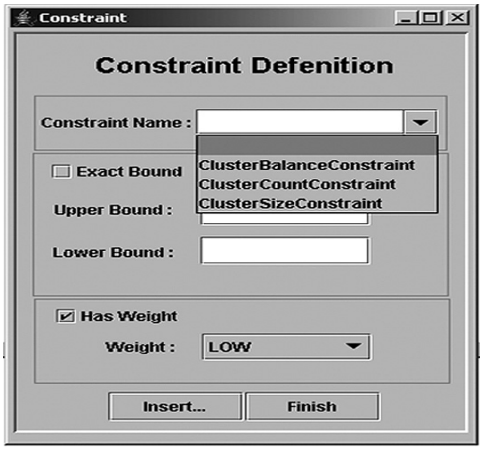


Figure 4: Selecting a constraint to be involved in the clustering algorithm

1. Minimum and maximum numbers of clusters,
2. Two or more nodes share a same cluster,
3. Limits on the variation in cluster size,
4. Bounds on the number of nodes in each cluster,
5. Two or more nodes appear in different clusters.

The first three constraints are defined as defaults in DAGC. The code for implementing these constraints and the results of applying the constraints is presented in the following subsections.

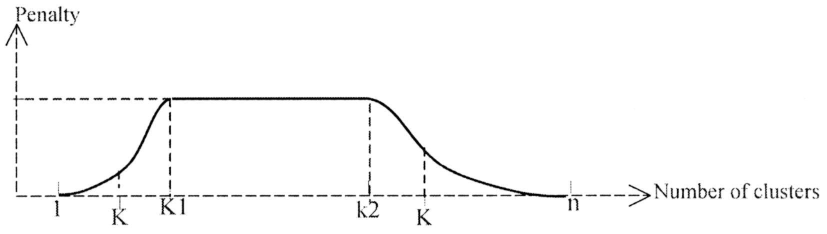
4.1 Limiting the Number of Clusters

In this subsection a constraint to restrict the number of clusters, k, to a range of integers, k1 to k2, such that $k1 \leq k \leq k2$, is presented. Since this constraint is applied to the number of clusters, the function g(c) should return the number of clusters for a given clustering, c. If for a given solution, c, the number of clusters, n, is within the range k1 to k2 then c is a feasible solution. Below, is the code for the functions g(c) and d(c):

```
Double  g(Clustering c)
    {return  c.clusterNumbers();  //returning the number of clusters }

    Double d(Clustering c)
    {if  g(c) >  k1 and g(c) <k2 then      // if c is feasible then  g(c) – g(cfeasible) is zero
        then return 0
         else
           return   min( |g(c)-k1| , |g(c) – k2|   )  // the least value for    g(c) – g(cfeasible)
    }
```

To alter the value of penalty, p, proportional to the number of feasible solutions in the genetic population the value of σ_t is recalculated for each generation using the relation 7 above. As shown in Figure 5, within the range of acceptable number of clusters, k1 to k2, the value of the penalty function, P, is 1. The value of P drops radically as the number of clusters approaches the minimum possible number of clusters, 1, and maximum number of clusters, n.



$$P(c) = (u\_p(c) - u\_p(c_0)) \;/\; (u\_p(c_f) - u\_p(c_0))$$

Figure 5: The penalty function curve for confining the number of clusters between k1 and k2

4.2 Balanced Cluster Sizes

The second default constraint is concerned with balancing the cluster sizes. That is, the graph nodes have to be equally partitioned into clusters of the same size. In order to impose the constraint, the method g(c) is implemented as follows:

```
Double  g(Clustering c)
{ avg = c.GraphNodeNumber() / c.clusterNumber()   // number of nodes in each
                                                   //cluster when clusters balanced
   // compute deviation from balance
  For i=1 to c.clusterNumbers()  do
        $\sigma^2$   +=  (c.clusters[i].size - avg )$^2$
   return $\sigma$;
}
```

In the above implementation of the function g, first the number of nodes in each cluster, when balanced, is computed and stored in a variable, avg. Then, the variance for the given solution, c, is computed and kept in σ^2. The mean deviation σ, is finally returned as the constraint value for c. The distance of the solution c from the feasible region, is defined as the mean deviation, σ, as well. Below, is the implementation for the method d(c):

```
Double d(Clustering c)
{ return g(c);      //since  g($c_{feasible}$) is zero so  g(c) – g($c_{feasible}$) is equal to g(c) }
```

4.3 Nodes Sharing a Same Cluster

There are certain situations where a number of graph nodes are to appear in the same cluster. Here, the constraint value, g(c), for a given solution c, is computed as the minimum number of movements required to move the specified nodes into a same cluster. Below, the methods g(c) and d(c) are presented:

```
Double  g(Clustering c)
{   Let  M={ni1,ni2,…,nit}  be the set of t nodes to be collocated
     Let tmax= maximum number of collocated nodes which belong to M.
     Movements = t - tmax  // minimum number of movements required
     Return  movements
}
Double  d(Clustering c) { return  g(c)  }
```

As described above in Section 2.1, the return value of the method g is used in relation (1) to calculate the value for un-scaled penalty value.

4.4 Experimental Results

In this section the quality of clustering and the number of feasible solutions in each generation of the genetic clustering algorithms when applying the normal distribution penalty function and a quadratic function are compared. The quadratic and normal distribution penalty functions are defined and implemented for the three constraints described above in Sections 4.1 to 4.3. The penalty functions are associated with a genetic clustering algorithm within the DAGC environment. The clustering algorithm is applied to a class dependency graph extracted from a Java source code. The graph is clustered by applying the constraints. For each constraint, the quality of the best feasible solution is recorded first with our adaptive normal distribution penalty function and then with the quadratic penalty function. The clustering objective function here is TurboMQ (Mitchell and Mancordis, 1999) which attempts to maximize cohesion of the nodes in each cluster and minimize the coupling amongst clusters.

As shown in Figure 6, by applying our adaptive normal distribution penalty function, higher quality of results for the three constraints can be achieved in comparison with the traditional quadratic penalty approach. A main reason is the adaptive nature of the normal distribution penalty function. In early generations of the genetic algorithm execution, the value of the parameter σ gets bigger because of the large amount of infeasible solutions in the population. As shown in Figure 1, the bigger the value of σ the less the curvature of the normal distribution function will be. Therefore, all the solutions in the population will be penalized almost identically and solutions with higher fitness will survive in the subsequent generation. By growing the number of feasible solutions gradually, the value of σ is reduced due to the higher ratio of feasible solutions in the population and the infeasible solutions will be penalized much more than before and consequently the algorithm tends to maintain the feasibility of the population.

Using the adaptive normal distribution penalty function not only produces feasible solutions with higher quality values but also finds feasible solutions earlier. Furthermore, in subsequent generations the amount of feasible solutions gets higher. Figure 7 shows a comparison of the degree of constraint violation when applying the normal distribution and the quadratic penalty function for

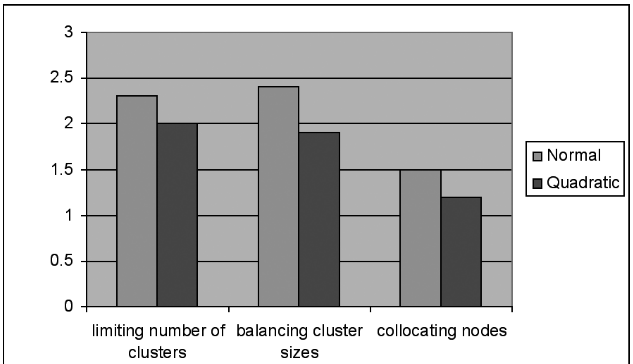
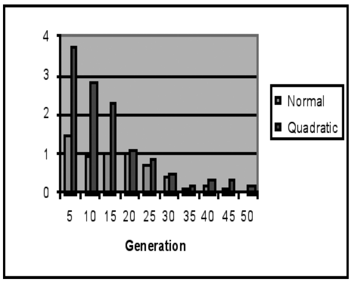
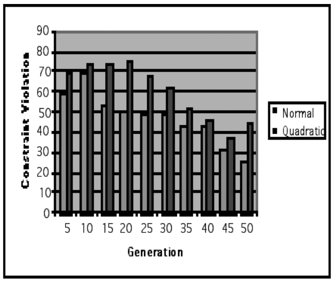
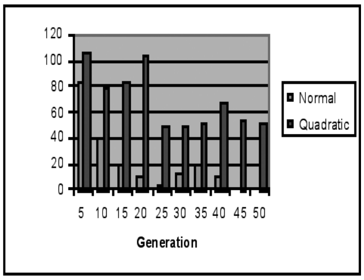


Figure 6: Qualities of results using normal distribution and quadratic penalty functions







(1) No. Nodes within a range **(2) Clusters of equal size** **(3) Nodes collocation**

Figure 7: Degree of feasibility for different generations

the three constraints. The value of constraint violation is a measure of the generation feasibility which is computed by summing up the amount of infeasibility of individuals in the generation.

5. GENERAL CONSTRAINED CLUSTERING ALGORITHM

In this section the design of the DAGC environment is described. In Section 5.1 a general constrained genetic clustering algorithm is presented. Section 5.2 includes a brief description of the DAGC architecture.

5.1 A General Constrained Clustering Algorithm

In order to achieve flexibility in modification and assembly of genetic clustering algorithms a general algorithm is presented. The algorithm invokes a basic set of operations, generally appearing in genetic clustering algorithms. The basic operations of a typical genetic algorithm are summarized below:

1. Map the search space of all possible solutions of the problem onto a set of finite strings (chromosomes) over a finite alphabet.
2. Randomly select the initial population (first generation) of solutions.
3. Compute the fitness (measure of the quality) of each individual in the population.
4. Perform crossover between pairs of individuals to create new individuals and replace the randomly-selected individuals with these new individuals.
5. Randomly mutate a small part of the resulting population from last steps.
6. Repeat the optimization process starting at point 3 until a stopping condition is met. For instance, the optimization process may stop when the population converges to a state where majority of the chromosomes represent a single solution which is considered to be best.

Typical operators for all types of genetic algorithms are: selection, crossover and mutation. Each of the operations can be implemented as a function with a standard interface, being accessed by a general clustering algorithm. To define a general algorithm a comprehensive study of the existing algorithms was carried out. In general, there are a fixed set of operators and GA parts appearing in genetic clustering algorithms which are highlighted in bold in Figure 8, opposite.

Defining, a function with a standard signature for each of the GA parts, invoked by the general algorithm, various implementations for the parts may be used, without any need to change the body of the algorithm. For instance, there are several schemes for the selection operator such as roulette wheel, tournament, elitist models, and ranking methods (Gen and Chang, 1997). Defining a unique

```
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. While not StopCondition() do
      for each Chromosome in current population
4.      decode(Chromosome)
5.      for each constraint Ci
6.         Compute P[i] = penalty(Chromosome)
7.       end for
8.       Calculate fitness(Chromosome,P)
       end for

9.    Create intermediate generation using Selection
10.     Crossover (c1, c2) resulting O1 and O2
                                   11.    Mutate O1 , O2
                  12.    Create Next Generation using Reproduce
   End While
```

Figure 8.: A general constrained clustering algorithm

Genetic Component	Interface	Description
Decode a chromosome	Interface Decoder{ Decode (Chromosome C, Graph G Vector NodeSequence);}	decodes a chromosome C into a clustering G.
Fitness Calculation	Interface Fitness{ CalculateFitness(Graph Chromosome C); }	Calculate the fitness of chromosome C.
CrossOver	Interface Xmethod{ CrossOver(Chromosome parrent_1, Chromosome parrent_2, Chromosome offspring_1, Chromosome offspring_2, double probability); }	Creating offspring from parents
Mutation	Interface mutation{ Domutation(Chromosome C, double probability); }	Mutate a chromosome c
Selection	Interface selection{ Select(Population thepop);}	Returns a chromosome according to the selection mechanism.
Local Improvement	Interface improvement{ Improve (Chromosome C, Decoder thedecoder,Graph g, Fitness thefitness); }	Returns the best neighbor of a chromosome if found.
Random Generator	Interface randoize{ randomize(Chromosome c); }	Generates the first generation population
Constraint	Interface Constraint{ Double g(Graph g); Double d(Graph g) ; Double d0(Graph g); Double sigma(Population thepop) ;}	A user defined constraint

Figure 9.: Interface definition for some of the genetic parts in DAGC

interface for the selection function each of these schemes may be used without modifying the rest of the genetic algorithm which makes use of the selection function. Below, the interfaces of some of the function implementations for the GA parts are presented.

Using the standard Interfaces presented in Figure 9, users can provide their own implementation of the genetic parts. For instance, user can define several constraints for the clustering genetic algorithm by implementing the interface described in Section 2.2.

5.2 DAGC Architecture

The architecture of DAGC is shown in Figure 10. There are two layers; the top layer consists of services provided by the tool. The bottom layer contains a complete programming API which implements the main functionality of DAGC.

5.2.1 API Layer

The API layer is an independent layer which includes a useful and complete set of Java classes. These classes are arranged in five packages as follows:

(1) ClusteringAPI: The ClusteringAPI package provides useful classes and interfaces which implements standard interfaces described in Section 3.

(2) 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 (Mitchell, 2002). These three types of graphs are used for assessment of the genetic algorithms developed within the DAGC environment. Random Graphs are shown by R.n.d symbol (Bui and Moon, 1996). Here, n is the number of graph nodes and d is the average degree of graph nodes (for example R.30.2 defines a graph with 30 nodes and average degree 2 for graph nodes). Software graphs are special types of Random Graphs. Experiments with several software graphs show that in these graphs the number of edges grows linearly with respect to the number of nodes, (O(|E|)=O(|V|)) (Mitchell, 2002). A random software graph is defined by two parameters: n which is the number of graph nodes

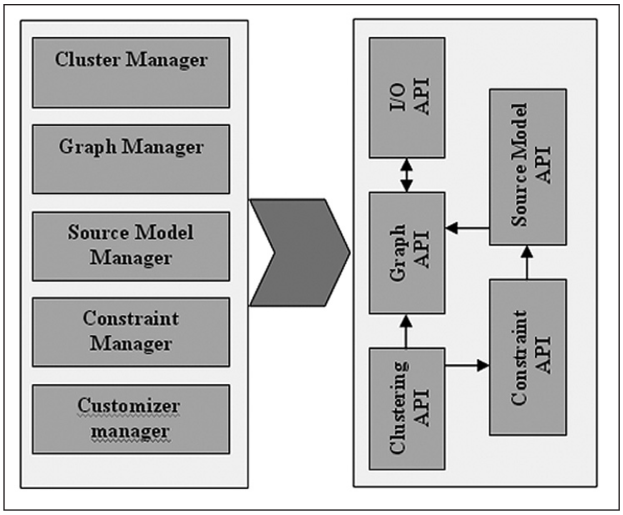


Figure 10: DAGC Architecture

and k, a coefficient which defines the proportion of edges and nodes of that graph. By creating a random software graph in DAGC, one can simulate a graph extracted from a real software system, so he will be able to produce a software-like graph immediately in DAGC to evaluate his GA.

(3) 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).

(4) Source Model API: This package contains a number of classes to extract call graphs from a Java source code. To extract class dependency graphs different algorithms (Sundaresan and Hendren, 2000) are available in the DAGC tool. These algorithms present different heuristics to resolve polymorphic calls. As we mentioned earlier, a call graph which is an input to the clustering algorithm, is a graph in which a node represents a class and an edge represents a call between two classes.

(5) Constraint API: This package contains classes and interfaces which provide predefined constraints described in the previous section and abstract interfaces allowing the user to augment new constraints into the environment.

5.2.2 Services Layer

This layer provides five major services through components called managers: Clustering Manager which allow the user to assemble a new genetic clustering and start that, Graph Manager which allow the user to make different benchmark graphs, Source Model Manager which is used for extracting dependency graphs from the Java source, Constraint manager which allows building of a constraint based clustering and Customizer manager which allows to augment user-defined components to the environment.

6. CONCLUSIONS

It is possible to simply apply constraints by implementing a general interface, presented in this paper. In order to implement the interface, a function to estimate the distance of each solution from the feasible region has to be supplied, by the user. Three implementations of the interface for applying the three common clustering constraints to limit the number of clusters within a given range, balance cluster sizes and collocation of certain nodes in a shared cluster is presented. The constraints are also implemented using a quadratic penalty function and compared with the normal distribution. Experimental results with the penalty functions show that the normal distribution functions eventually leads the genetic clustering algorithm to the feasible region faster than the quadratic penalty functions.

The normal distribution penalty function is adaptive by its nature because, the variance of the distribution depends on the proportion of feasible solutions in a population. Hence, as the number of feasible solutions in a genetic population increases the variance reduces and the skewness of the normal penalty function increases. As a result, more penalty will be applied to less feasible solutions. Our experimental results demonstrate a better quality of the solutions when applying the normal distribution compared with the quadratic penalty functions.

Providing a standard interface for the constraint supports defining any arbitrary user-defined constraint and augmenting it to the environment. Imposing constraints on the genetic clustering always reduces the quality of solutions in the population. By using the normal distribution penalty function less reduction in the quality values of the feasible solutions was observed.

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.C. and LETHBRIDGE T. (1999): Experiments with clustering as software re-modularization method, in Proc. of working Conf. On Reverse Engineering, *IEEE* October.

BUI, T. N. and MOON, B. R. (1996): Genetic algorithm and graph partitioning, *IEEE Trans. Comput*., 45: 841–855, July.

COELLO, C. (2002): Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: A survey of the state of the art, *Computer Methods in Applied Mechanics and Engineering*, 191: 1245–1287.

CROSSLEY, W.A. and WILLIAMS, E.A. (1997): A study of adaptive penalty functions for constrained genetic algorithm based optimization, Aerospace Sciences Meeting and Exhibit, 35th, Reno, NV, January 6–9.

GEN, M. and CHANG, R. (1997): Genetic algorithm and engineering design, John Wiley and Sons.

HOMAIFFAR, A., QI, C. and LAI, S. (1994): Constrained optimization via genetic algorithms, *Simulation*, 62: 242–254.

JOINES, J. and HOUCK, C. (1994): On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with GAs, *Proceedings of the First IEEE Conference on Evolutionary Computing*, 579–584.

KIMELMAN, D., ROTH, T., LINDSEY, H. and THOMAS, S. (1997): A tool for partitioning distributed object applications based on communication dynamics and visual feedback, Proceedings of the Advanced Technology Workshop, *Third USENIX Conference on Object-Oriented Technologies and System*.

LOPEZ-VALLEJO, M.L., GRAJAL, J. and LOPEZ, J.C. (2000): Constraint-driven system partitioning, Design, Automation and Test in Europe (DATE '00).

MITCHELL, S.B. (2002): A heuristic search approach to solving the software clustering problem, Thesis , Drexel University, March.

MITCHELL, S.B. 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*.

PARSA, S. and BUSHEHRIAN, O. (2004): A framework to investigate and evaluate genetic clustering algorithm for software re-modularization, *Lecture Notes in Computer Science*, 3036.

PARSA, S. and BUSHEHRIAN, O. (2005): A new encoding scheme and a framework for investigating genetic clustering algorithms, *The Journal of Research and Practice in Information Technology*, 37(1): 127–143.

PARSA, S. and BUSHEHRIAN, O. (2002): The design and implementation of a framework to automatic modularization of software systems, *Journal of Super Computing*, 31, April.

SUNDARESAN, V. and HENDREN, L. (2000): Practical virtual method call for Java, *Proceedings of the conference on Object-Oriented Programming, Systems, Languages, and Applications*.

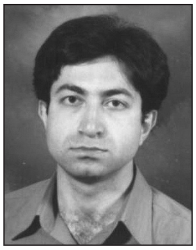
TILEVICH, E. and SMARAGDAKIS, Y. (2002): J-rchestra:Automatic Java application partitioning, *European Conference on Object-Oriented Programming (ECOOP)*, Malaga, June.

BIOGRAPHICAL NOTES

Saeed Parsa received the BSc in mathematics and computer science from Sharif University of Technology, Iran, the MSc degree in computer science and the 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 the BSc in software engineering from AmirKabir University of Technology (Tehran Polytechnics), Iran, the MSc degree in software engineering from the 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