Finding Effective Software Metrics to Classify Maintainability Using a Parallel Genetic Algorithm

Rodrigo Vivanco[1,2] and Nicolino Pizzi[1,2]

[1]Institue for Biodiagnostics, National Research Council Canada, Winnipeg, MB, Canada
{Rodrigo.Vivanco, Nicolino.Pizzi}@nrc-cnrc.gc.ca
[2]University of Manitoba, Winnipeg, MB, Canada
{rvivanco, pizzi}@cs.umanitoba.ca

Abstract. The ability to predict the quality of a software object can be viewed as a classification problem, where software metrics are the features and expert quality rankings the class labels. Evolutionary computational techniques such as genetic algorithms can be used to find a subset of metrics that provide an optimal classification for the quality of software objects. Genetic algorithms are also parallelizable, in that the fitness function (how well a set of metrics can classify the software objects) can be calculated independently from other possible solutions. A manager-worker parallel version of a genetic algorithm to find optimal metrics has been implemented using MPI and tested on a Beowulf cluster resulting in an efficiency of 0.94. Such a speed-up facilitated using larger populations for longer generations. Sixty-four source code metrics from a 366 class Java-based biomedical data analysis program were used and resulted in classification accuracy of 78.4%.

1 Introduction

Software project managers commonly use various metrics to assist in the design and implementation of large software systems [1,2]. These metrics are used to quantify the various developmental stages of the project; design metrics such as the number of classes, level of inheritance, number of abstract classes and so on; implementation metrics obtained automatically from the source code, e.g., lines of codes per method, lines of comments, number of methods in a class, method complexity, number of tokens, and so on; test metrics such as the number of errors reported; and usability metrics like the number of times the user pressed the wrong button or asked for help. One important aspect of project management is the ability to identify potential problems in the design and development phases from the source code as the application is implemented. Project managers and developers can use source code metrics to model and predict the quality of the software as the application evolves.

The use of computational intelligence methods such as neuro-computing, fuzzy computing and evolutionary computing in software engineering is being recognized as an important tool for software engineers and project managers [3]. Due to the attributes of the problem, software engineering is stochastic in nature were the skill set and experience of developers and project leaders play a large factor in the final overall quality of the software. As such, search-based techniques such as evolutionary computing are drawing the attention of researchers to help develop and fine-tune software engineering tools [4].

The ability to evaluate the quality of a software object can be viewed as a classification problem. Given a set of objects (object-oriented classes), with known features (source code metrics) and class labels (expert quality rankings) build a classifier that is able to predict the quality of a software object from its metrics. Not all metrics have the same discriminatory power when it comes to predicting the quality of a software object. For example, the number of semicolons may not be as powerful as the number of lines of code, which may not be as powerful as a measure of the amount of coupling, cohesion and complexity in predicting the quality of a software object in terms of maintainability. Also, the combination of metrics may be more important than any individual metric, a previously unforeseen combination of metrics may capture the insight that an expert uses when attributing a quality ranking to a software object. Determining which metrics, and combination of metrics, have strong discriminatory powers is paramount for generating a good classifier. However, finding an effective combination of metrics that have good discriminatory properties is not a problem that can be solved analytically as the number of metrics increases.

Genetic algorithms have been used extensively to find feature subsets in classification problems in the biomedical field [5,6,7]. However, computational intelligence techniques have not been used as extensively in the software engineering domain. [8] used a genetic algorithm to find the optimum neural network architecture for a two-class problem and identify fault-prone modules using nine software metrics. [9] utilized a genetic algorithm approach to find a model that best classifies fault-prone modules based on 5 metrics. [10] exploited genetic algorithms to determine software stability, how much a Java class changed over the course of the project, using 11 metrics based on coupling, cohesion, inheritance and complexity.

The objective of this report is two fold. First, to illustrate how a genetic algorithm was used as an effective feature sub-selection strategy to a classification problem in the software engineering domain, that is, determine which subset of object-oriented source code metrics are able to predict the quality of a software object in terms of maintainability. Secondly, to show how a parallelized version of the canonical genetic algorithm was implemented using the Message Passing Interface (MPI) library [11]. The speed up of the program execution enabled more combinations of GA parameters, such as population size, mutation rate and number of generations to be tried within a reasonable amount of time for the researcher, which facilitated finding an optimal solution.

2 Software Metrics

All 366 software objects in an in-house Java-based biomedical image data analysis system, were subjectively labeled by an experienced software architect in terms of maintainability. The architect (8 years of programming experience, 7 years with object-oriented systems and 5 years with Java) was asked to rank each software object. That is, based on personal experience, assign a value from 1 to 5 that tries to rank the overall design and implementation of a particular software object. Software objects with low rankings are determined to be difficult to modify, and should be reviewed by the development team in efforts to improve the class, either by improving the design (which may mean refactoring classes) or improving the implementation of the methods. A class ranked 1 should definitely be subject to a review as soon as possible. A ranking of 2 meant the class indicated that the class should be reviewed but it is not critical that it be done immediately. A ranking of 3 indicates, in the opinion of the expert, an average design and implementation, neither exemplary or detrimental to product quality and maintenance. A class ranked 4 is better than average but could use some improvements (for example, better documentation). A class ranked 5 is considered very easy to understand and modify. The software architect was not instructed to focus on any particular aspects of code quality (in reference to metrics that can be measured) but to use his experience and intuition to rank the software classes. Table 1 shows the distribution of the labeled software objects by the expert familiar with the project in terms of design and implementation.

Table 1. Distribution of software object labels assigned by an expert

Rank 1	Rank 2	Rank 3	Rank 4	Rank5
2	56	75	94	139

The project has been in development for over 24 months and low ranking classes have been identified and corrected so few „must review“ software objects are present in the current dataset. This will bias the classifier towards properly identifying high ranking classes. The majority of the software classes ranked at level 5 are simple data-model classes with simple get/set methods, they basically encapsulate classes are not highly coupled to other classes and their method complexity are low.

A set of 64 metrics was obtained from an evaluation version of Borland's TogetherSoft package [12] and an in-house metrics parser. Due to the nature of the application, it was noted that data model objects were relatively simple when compared to graphical user interface (GUI) classes, so an additional metric value was used to aid the classifier, a distinction was made between GUI, data model, algorithm, and all other objects. In general, data model classes had many get/set methods and low coupling, while GUI classes had higher coupling among themselves and the data that was to be displayed. The software metrics that were used to classify the Java software objects are shown in Table 2.

Table 2. The 64 object-oriented source code metrics used as features for the classifier

Description	Metrics
TYPE	Type: GUI (=1), Data Model (=2), Algorithm (=3), Other (=4).
METH	# methods.
LOC	# lines of code.
ALOC	Mean LOC per method.
MLOC	Median LOC per method.
RCC1	Ratio of comment lines of code to total lines of code including white space and comments.
RCC2	TCR from Borland's TogetherSoft
TOK	# tokens.
ATOK	Mean TOK per method.
MTOK	Median TOK per method.

DEC	# decisions: for, while, if, switch, etc.
ADEC	Mean DEC per method.
MDEC	Median DEC per method.
WDC	Weighted # decisions based on nesting level *i*: Sum[$i*n_i$]
AWDC	Mean WDC per method.
MWDC	Median WDC per method.
INCL	# inner classes.
DINH	Depth of inheritance.
CHLD	# children.
SIBL	# siblings.
FACE	# implemented interfaces.
RCR	Code reuse: ratio of overloaded inherited methods to those that are not.
CBO	Coupling between objects.
LCOM	Lack of cohesion of methods.
RFO	Response for an object. Response set contains the methods that can be executed in response to a message being received by the object.
RFC	Response for class.
MNL1	Maximum method name length.
MNL2	Minimum method name length.
MNL3	Mean method name length.
MNL4	Median method name length.
ATCO	Attribute Complexity.
CYCO	Cyclomatic Complexity.
DAC	Data Abstraction Coupling.
FNOT	Fan Out.
HLDF	Halstead Difficulty.
HLEF	Halstead Effort.
HLPL	Halstead Program Length.
HLVC	Halstead Program Vocabulary.
HLVL	Halstead Program Volume.
HLON	Halstead # operands.
HLOR	Halstead # operators.
HLUN	Halstead # unique operands.
HLUR	Halstead # unique operators.
MIC	Method Invocation Coupling.
MAXL	Maximum # levels.
MAXP	Maximum # parameters.
MAXO	Maximum size operations.
ATTR	# attributes.
ADDM	# added methods.
CLAS	# classes.
CHCL	# child classes.
CONS	# constructors.
IMST	# import statements.
MEMB	# Members.
OPER	# operations.
OVRM	# overridden methods.
REMM	# remote methods.
PKGM	% package members.

PRVM	% private members.
PROM	% protected members.
PUBM	% public members.
DEMV	Violations of Demeters Law.
WMC1	Weighted Methods per class.
WMC2	Weighted methods per class.

3 Parallel Genetic Algorithm

Evolutionary computation algorithms such as genetic algorithms (GA) attempt to discover an optimal solution to a problem by simulating evolution [13]. In GA, a solution (set of software metrics) is encoded in a gene and a collection of genes (solutions) constitutes a population. GA uses natural selection and genetics as a basis to search for the optimal gene, a set of software metrics that give the best classification rate. A population of solutions is modified using directed random variations and a parent selection criteria in order to optimize the solution to a problem. They are based on the process of Darwinian evolution; over many generations, the „fittest" individuals tend to dominate the population.

The simplest way to represent a gene is to use a string of bits, where 0 means the bit is off and 1 means the bit is on. For this problem domain, the 64 metrics were encoded in a 64-bit string. A zero bit meant the metric was not to be used with the classifier. All the metrics with a corresponding bit set to one constituted the metrics sub-set to evaluate with the fitness function, the classifier. The genes of the initial population were randomly initialized.

To evaluate a gene's fitness a linear discriminant analysis (LDA) classifier was utilized using the leave-one-out method of training and testing. LDA is a conventional classifier strategy used to determine linear decision boundaries between groups while taking into account between-group and within-group variances [14]. If the error distributions for each group are the same (each group has identical covariance matrices and sampled from a normal population), it can be shown that linear discriminant analysis constructs the optimal linear decision boundary between groups. Figure 1 shows a three-class 2-dimensional classification problem and the decision hyper-planes produced by LDA. For the leave-one-out validation method, 365 classes were used to train the LDA, and then the software object left out was tested with classifier. This was repeated for all 366 software objects. The classification rate was the number of times the software object left out to be tested was correctly classified by the LDA. The final classification rate for a feature subset is a value between 0 and 100%.

The genes with higher fitness values are more likely to be chosen for reproduction and evolve into the next generation. For the implemented GA, a random probability was generated, and a random gene with an equal or larger fitness value was chosen as a parent. All the genes in the population were candidate parents. A child gene is created by picking a crossover point and exchanging the corresponding bits from the two parent genes. A single crossover point was randomly chosen for the creation of the child gene. Mutation was performed by flipping the value of each bit in the child

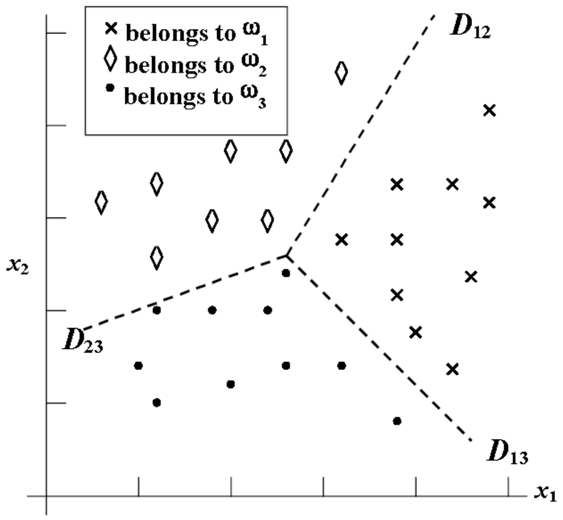

Fig. 1. Possible linear discriminant analysis decision boundaries for three groups

gene if the random probability value is greater than 1.0 - P, P being the user specified mutation probability parameter. The number of genes in the population is also a user set parameter and did not change during successive generations.

A new population is created by merging the children with the elite genes of the previous population. The number of elite genes is a parameter set by the user. The new population is sorted based on fitness values and some of the children may now be in the elite pool. The elite pool is only used for merging the next generation, as the entire population is eligible for reproduction.

The canonical genetic algorithm is an ideal candidate for coarse grain domain partition parallelization in an effort to improve the computational speed for finding a solution to a non-deterministic problem [15]. The fitness function for a solution gene can be executed independently of the other genes in the population. That means one process can calculate the fitness function for one gene, while another process does the same for another gene in parallel. After the current children population is evaluated, the previous generation and current generation are merged. The most computationally intensive function for the canonical GA is usually the calculation of a gene's fitness value. Figure 2 illustrates the basic modification of the sequential GA to make it parallelizable using a manager/worker approach.

The sequential and parallel versions of the canonical genetic algorithm were implemented using the C++ language. The MPI library which facilitates the message passing of data between processes in a parallel computer or a cluster of computers using C-style function calls. It is a well-established protocol and widely implemented on various operating systems. The parallel GA program was tested on a 20 node heterogeneous Beowulf cluster running the Linux operating system. For the sequential GA program the population was set to 100, the percent elite to 25%, mutation rate at 5% and the number of generations capped to 150. For the parallel version, which

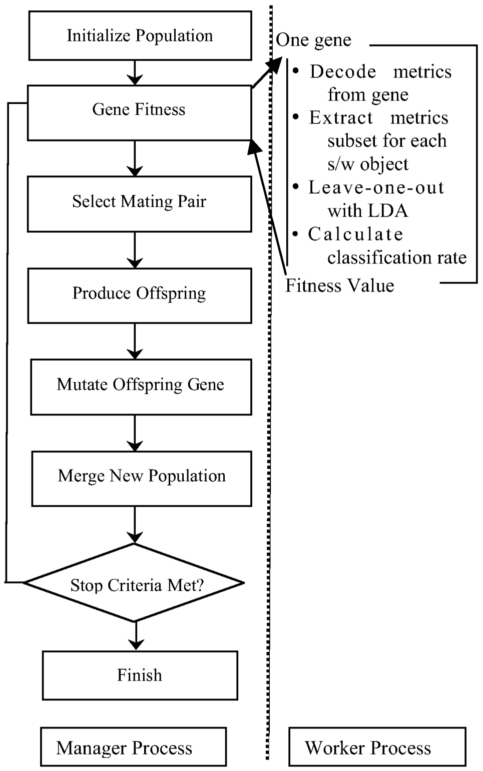

Fig. 2. Manager/worker parallel version of the canonical sequential genetic algorithm

executed in a significantly shorter amount of time, a range of population sizes were tested for various generations, with the same mutation rate and percent elite as the sequential GA.

Using a manager/worker paradigm, the parallel section of the algorithm is executed by a different worker process while the main tasks of merging the parent and children population, creating new genes and testing for the termination condition is done by the manager process. In the method where the population fitness is calculated, the manager process repeatedly receives a fitness value from a worker process, and if there are genes still to be evaluated, sends one to the idle worker process. Below is the pseudo code for method in the manager that calculates the fitness for all the genes in a population.

```
void calculate_population_fitness( population ) {
  num_workers = number of processes in MPI network
  //initialize workers, give each one a gene to process
  num_genes_evaluated = 0
  for (worker_id=0;worker_id<num_workers;worker_id++){
     // send a gene, and matching gene_id, to a worker
     MPI_Send( gene_bits, gene_id, worker_id )
```

```
      num_genes_evaluated += 1
    }
    // while there is work left, get fitness value from a
    // worker, give the worker another gene to evaluate
    while ( num_genes_evaluated < population_size ) {
      // get fitness from a worker
      MPI_Receive( fitness, gene_id, worker_id )
      set fitness in population using gene_id
      // send another gene to the worker
      MPI_Send( gene_bits, gene_id, worker_id )
      num_genes_evaluated += 1
    }
    // no more genes to evaluate, collect remaining
    // results from workers
    for (worker_id=0;worker_id<=num_workers;worker_id++){
      MPI_Receive( fitness, gene_id, worker_id )
      set fitness in population using gene_id
    }
  } // end of method
```

The worker process decodes the gene and generates a new metrics dataset. For all the software objects, only the metrics with a corresponding bit set to one are used with the classifier. Using LDA with the leave-one-out train/test technique the classification rate using the metrics subset encoded in the gene is returned as the fitness value. Once all the genes in a population are evaluated, the manager process continues with the canonical GA in a sequential manner. The algorithm is repeated until the specified number of generations is executed.

4 Results and Observations

The parallel GA program runs substantially faster than the sequential version. It took 14.5 hours to run a 100 gene population for 150 generations with the sequential program, running on a 1.6 Ghz CPU with 2 Gigs of RAM. Using the same GA parameters with the parallel version on the Beowulf cluster took 1.3 hours. With such a speed up more variations of GA parameters could be tried. Figure 3 shows the classification rate when all the available metrics were used with the classifier, the best classification from a set of 100 random metrics subsets, and the best classification when different generations were used with 100 genes, 5% mutation rate and 25% elite genes. Modifying the percent elite and mutation rates did not result in noticeable better classification rates, though increasing the size of the population did. An improved classification rate was obtained when the number of generations was increased.

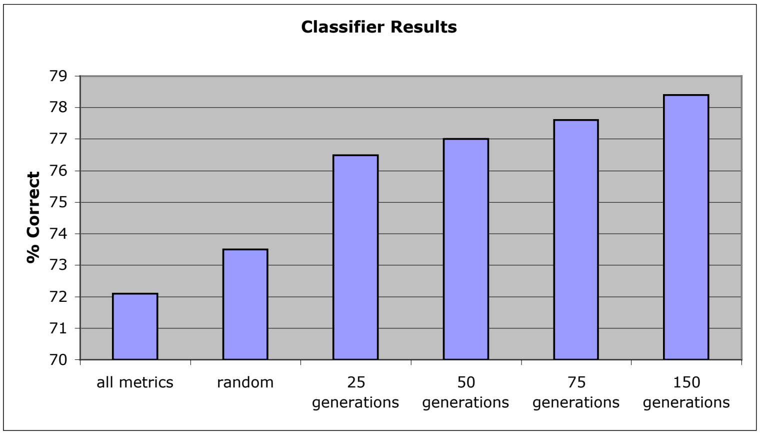

Fig. 3. Classification rates when all metrics are used, random subsets, and GA feature selection

Table 3 shows the metrics encoded by the top 3 genes, they provided the best classification for the software classes with a percent correct of 78.4%, 78.1% and 77.9%. Different genes encode slightly different metrics yet result in comparable classifier performance. It is interesting to note the metrics that were common to all 3 genes. ALOC, the mean number of lines would seem to indicate that the size of a method affects maintenance, which intuitively makes sense, along with other complexity metrics such as HLDF. One of the software objects that was ranked as must review had over 6,000 lines of code. It was commented that this class seemed like an application onto itself. It was the main data visualization GUI class for the application from which the user could select further processing options.

The MNL1 metric, method name length, would suggest that using longer names for methods (and variables, though that is not a metric used in this study) facilitates the understanding of the source code by other programmers, and thus affects the maintainability of the code, since longer labels names tend to indicate more meaningful names which express the purpose of the method and the variable. The other method name length metrics, such as MNL2, MNL3 and MNL4 are variations of the same information captured by the MNL1 metric. The other classes ranked 1 had 900 lines of code but was very poorly documented and highly coupled to other classes making the purpose of the interdependencies not clear to the reviewer.

Coupling measures were also encoded in the top genes. Indirect measures such as package members, PKGM, indicate potentially shared data members among classes. Inheritance measures CHCL (number of children) and FNOT (fan out) are also present in the metrics subset. Only the top gene explicitly used the coupling CBO metric from the CK metrics suite [16]. The only comments based metric, RCC1, was not included in the top three genes though it was included in other genes that performed well (77.6%). It is generally accepted that comments aid in the maintenance of software, and if more comment based metrics would have been generated the chances of top performing genes including that metric would have increased.

Table 3. Metrics encoded by top 3 genes and corresponding classification rates

Description	78.4%	78.1%	77.9%
TYPE	YES		
LOC	YES		YES
ALOC	YES	YES	YES
TOK			YES
ATOK	YES		YES
MTOK		YES	YES
DEC	YES		
ADEC		YES	YES
MDEC		YES	YES
WDC		YES	
AWDC	YES	YES	
DINH	YES	YES	
CHLD	YES		
FACE	YES	YES	
RCR	YES		
CBO	YES		
LCOM		YES	YES
MNL1	YES	YES	YES
MNL3		YES	
CYCO	YES	YES	
FNOT		YES	
HLDF	YES	YES	YES
HLEF	YES		
HLPL	YES	YES	YES
HLVL	YES	YES	YES
HLON		YES	
HLUN		YES	YES
MIC			YES
MAXL			YES
MAXP	YES		YES
MAXO	YES	YES	YES
ATTR	YES	YES	YES
ADDM		YES	YES
CLAS			YES
CHCL			YES
MEMB	YES		
OPER		YES	YES
OVRM	YES	YES	
PKGM	YES	YES	YES
PRVM	YES	YES	
PROM	YES	YES	YES
PUBM	YES	YES	
DEMV	YES		
WMC1	YES	YES	
WMC2		YES	YES

Figure 3 shows that using a subset of the metrics helps the classifier achieve better performance, as using a random subset results in improved performance over using all the available metrics, some of which capture redundant information about the source code. Using a directed search like a genetic algorithm enhances the classifier performance even further, and the longer the search the better the classification rate. Using a parallel genetic algorithm significantly reduces the computational time of the algorithm and facilitates longer searches. The resulting metrics subset that capture the intuitive knowledge of the expert can be further inspected and analyzed from a theoretical aspect to understand what aspects of design and implementation lead towards high quality software objects.

The initial class labels were subjectively assigned by the expert, hence the metrics of the genes that generate the best classifier will tend to reflect the aspects of the source code that the expert may intuitively deem important for distinguishing a class that is well designed and coded. On a production version of this approach, various experts knowledgeable with the problem domain and organization would label the software objects, priming the classifier to use the metrics that a particular organization deems critical in identifying problem software classes. Another area of future research would be to reduce the number of classes. Though a pass/fail labeling may not work so well, as there will tend to be many more pass software objects. A three class labeling scheme may be more appropriate. Using an in-house project does not lead to reproducibility of the method, or the comparison of using alternate methods by other researchers as in-house code is not usually available for distribution. Choosing an open-source project as a dataset would be a workable alternative and may lead to more collaborative research efforts.

Acknowledgement. The Natural Sciences and Engineering Council (NSERC) is gratefully acknowledged for its support of this investigation.

References

[1] Kan S.H.: Metrics and Models in Software Quality Engineering. Addison-Wesley Publishing Company, Reading, Massachusetts, USA (1995)

[2] Fenton N.E., Pfleeger S.L.: Software Metrics: A Rigorous and Practical Approach, 2nd Edition. PWS Publishing Company, Boston, USA (1997)

[3] Pedryc W.: Proc. ACM Software Engineering Knowledge Engineering. Computational Intelligence as an Emerging Paradigm of Software Engineering. Ischia, Italy (2002) 7-14

[4] Harman M., Jones B.F.: ACM SIGSOFT Software Engineering Notes. The SEMINAL Workshop: Reformulating Software Engineering as a Metaheuristic Search Problem, Vol. 26 (2001) 62-66

[5] Nikulin A.E., Dolenko B., Bezabeh T., Somorjai R.J.: NMR Biomed. Near-optimal feature selection for feature space reduction: novel preprocessing methods for classifying MR spectra, Vol. 11 (1998) 209-216

[6] Yang, J., Honavar V.: IEEE Intelligent Systems. Feature subset selection using a genetic algorithm, Vol. 13 (1998) 44-49

[7] Raymer M.L., Punch W.F., et al.: IEEE Trans on Evolutionary Computation. Dimensionality reduction using genetic algorithms. Vol. 4 (2000) 164-171

[8] Hochman R., Khoshgoftaar T.M., Allen A.B., Hudepohl J.P.: Proc. 7th IEEE Int. Symposium on Software Reliability Engineering. Using the Genetic Algorithm to Build Optimal Neural Networks for Fault-Prone Module Detection, White Plains, New York (1996) 152-162

[9] Liu Y., Khoshgoftaar. T.M. Proc. 6th IEEE International Symposium on High Assurance Systems Engineering. Genetic Programming Model for Software Quality Classification (2001)

[10] Azar D., Precup D., Bouktif S., Kegl B., Sahraoui H.: Proc. 17th IEEE International Conference on Automated Software Engineering, Combining And Adapting Software Quality Predictive Models by Genetic Algorithms (2002)

[11] Message Passing Interface (MPI) Standard, http://www-unix.mcs.anl.gov/mpi/

[12] Borland's TogetherSoft, http://www.borland.com/together.

[13] Goldberg D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading, Massachusetts, USA (1997)

[14] Duda, C.D., Hart P.E., Stork D.G.: Pattern Classification. Wiley & Sons, New York, USA (2001)

[15] Cantu-Paz E.: Effective and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Boston, USA (2000)

[16] Chidamber SR, Kemerer CF.: IEEE Trans on Software Engineering, .A Metrics Suite for Object Oriented Design, Vol. 20 (1994) 476-493