Rt
00594
c.1

A HYBRID HEURISTIC APPROACH TO OPTIMIZE RULE BASED SOFTWARE QUALITY ESTIMATION MODELS

Rita Korkmaz

B.S., Computer Science, Lebanese American University, 2006

Thesis submitted in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science

Division of Computer Science and Mathematics

LEBANESE AMERICAN UNIVERSITY

June 2008

LEBANESE AMERICAN UNIVERSITY

Thesis approval Form (Annex III)

Student Name: Rita Korkmaz I.D. #: 200301473

Thesis Title : A Hybrid Heuristic Approach to Optimize Rule-Based Software Quality Estimation Models

Program : Computer Science

Division/Dept : Computer Science and Mathematics

School : **School of Arts and Sciences**

Approved by:

Thesis Advisor: Dr. Danielle Azar

Member : Dr. Haidar Harmanani

Member : Dr. Chadi Nour

Member : —

Date 4/6/2008

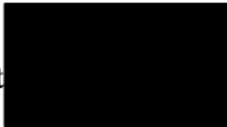
(This document will constitute the first page of the Thesis)

Plagiarism Policy Compliance Statement

I certify that I have read and understood LAU's Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.

This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Rita Korkmaz

Signat  Date: June 5, 2008

I grant to the LEBANESE AMERCIAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or its students and employees. I further agree that the University may reproduce and provide single copies of the work to the public for the cost of reproduction.

Acknowledgments

To begin with, I would like to strongly thank my supervisor and mentor Prof. Danielle Azar for her unconditional help, continuous encouragement and unshakable support. I especially thank her for introducing me to the software quality estimation field and proposing the interesting idea of developing hybrid approaches for the optimization of software quality estimation models. Facing my apprehensions and concerns with a smile and supportive words, Dr. Azar always managed to cheer me up and incite me to surpass myself. Source of inspiration, guidance and high quality demand, she has encouraged me to pursue the hard work and always strive to achieve superior results. I would also like to thank Dr. Haidar Harmanani, Chairman of Computer Science, for his general guidance and support throughout all my college years at LAU. I appreciated his confidence in my capabilities as well as his tutorship for some of the challenging academic assignments. Moreover, I would like to express my gratitude to the academic staff and general administration at LAU for making my five years stay a beneficial and fruitful as well as pleasant learning experience.

Special thanks and recognition to the CNRS (Centre National de Recherche Scientifique) which funded the basic research and experimentation required for this thesis. I also thank Jihane Andraos, Michel Barakat as well as lab personnel for assisting me in running the experiments in the labs.

Finally, special thoughts of appreciation to my parents for their unwavering support, both morally and financially. Their encouragement and patience were instrumental in allowing me to achieve my goals. Once again, a final word of gratitude to Professors Azar, Harmanani and Nour for their mentorship and participation in evaluating this thesis.

Abstract

Software quality is defined as the degree to which a software component or system meets specified requirements and specifications. Objectively quantifying software quality is important as it reduces cost, effort and time. Nonetheless, assessment of software quality is a difficult task since most software quality characteristics such as maintainability, reliability and reusability cannot be directly measured. However, they can be predicted from other software quality attributes such as complexity and inheritance. Many metrics to measure these software quality attributes have been proposed. These metrics are used to build software quality estimation models that take different forms: statistical models, rule based models and decision trees. In our work, we will deal with rule based estimation models for two main reasons: First, their white box nature makes them easy to interpret by human experts. Second, they supply guidelines that clearly show how to reach the prediction. Since data used to build such estimation models is scarce in the software quality estimation field, the accuracy of these rule based models gravely deteriorates when applied on new unseen data. The goal of this thesis is to explore and assess the use of hybrid heuristics to improve and adapt the rule based models to the context specific datasets.

Table of Contents

1 INTRODUCTION **5**

1.1 DEFINING SOFTWARE QUALITY 6

1.1.1 Software Quality Characteristics 6

1.1.2 Software Quality Attributes and Metrics 7

1.2 SOFTWARE QUALITY ESTIMATION MODELS 9

1.3 PROBLEM STATEMENT AND OBJECTIVE 11

1.4 THESIS ORGANIZATION 12

2 BACKGROUND **13**

2.1 INDUCTIVE LEARNING BACKGROUND. 13

2.1.1 Classification Models 15

2.1.1.1 C4.5 Algorithm 15

2.1.1.2 Decision Trees 17

2.1.1.3 Rule Sets 18

2.1.2 Evaluation Criteria 19

2.2 PREVIOUS AND RELATED WORK 22

2.2.1 Previous Work in Building Software Quality Estimation Models 22

2.2.1.1 Previous Work in Building Statistical Software Quality Estimation Models 22

2.2.1.2 Previous Work in Building Logical Software Quality Estimation Models 23

2.2.2 Related Work in Optimizing Existing Rule Based Software Quality Estimation Models 26

3 GENETIC ALGORITHMS **28**

3.1 THE DARWINIAN THEORY 28

3.2 THE ALGORITHM 30

3.2.1 Encoding Schemes 31

3.2.2 Genetic Operators 32

3.2.2.1 Crossover 32

3.2.2.2 Mutation 34

3.2.2.3 Elitism 35

3.2.3 Selection Schemes 35

3.2.3.1 Roulette Wheel Selection 36

3.2.3.2 Rank Selection 36

3.2.3.3 Tournament Selection 37

3.2.4 Replacement Policies 38

3.2.5 Termination Conditions *38*

4 SIMULATED ANNEALING **39**

4.1 THE METROPOLIS ALGORITHM: BOLTZMANN SAMPLING AND MONTE CARLO SIMULATION. 39

4.1.1 Annealing: a thermal physical process. *39*

4.1.2 The Metropolis Monte Carlo Algorithm *40*

4.1.3 Boltzmann distribution *41*

4.2 THE SIMULATED ANNEALING ALGORITHM 41

5 TABU SEARCH **44**

5.1 BACKGROUND 44

5.2 THE TABU SEARCH ALGORITHM 45

5.2.1 Elements of Tabu Search *45*

5.2.2 The algorithm *46*

5.2.3 Intensification and diversification *47*

6 THE HYBRID HEURISTICS *DESIGN AND EXPERIMENTAL RESULTS* **48**

6.1 THE HYBRID HEURISTICS: ALGORITHMIC DESIGN 48

6.1.1 The general algorithm *48*

6.1.2 The Genetic Algorithm *51*

6.1.2.1 The graph representation 51

6.1.2.2 The selection scheme 53

6.1.2.3 The fitness function 53

6.1.2.4 The genetic operators 54

6.1.2.5 Post-Processing and Trimming 67

6.1.2.6 The termination condition 68

6.1.3 The Simulated Annealing Algorithm *68*

6.1.3.1 The evaluation function 69

6.1.3.2 The perturbation function 69

6.1.4 The Tabu Search Algorithm *70*

6.1.4.1 The evaluation function: 70

6.1.4.2 The Neighborhood function: 70

6.1.4.3 The Tabu List Replacement Policy 71

6.2 THE EXPERIMENTS 72

6.2.1 The Datasets *72*

6.2.2 Experimental Settings *77*

6.2.3 Experimental Results and Analysis *82*

6.2.3.1 Experiment 1: Done on STAB2 using accuracy as the evaluation function. 82

6.2.3.2 Experiment 2: Done on STAB2 using accuracy + 0.5*J-Index as the evaluation function. 84
6.2.3.3 Experiment 3: Done on STAB1 using J-Index as the evaluation function 86
6.2.3.4 Comparison with results obtained in previous work 88
6.2.3.5 Limitations of the technique 90
6.2.3.6 Insight on the tri-phased optimization process of the hybrid heuristics 91
7 CONCLUSION AND FUTURE WORK 94

Table of Figures

Figure 1: A decision tree and its corresponding rule set. 16
Figure 2: A decision tree predicting the maintainability of class based on complexity metrics. 18
Figure 3: An example of single point crossover. 33
Figure 4: An illustration of double-point crossover. 34
Figure 5: An illustration of uniform crossover. 34
Figure 6: Single-bit mutation at position 6. 35
Figure 7: Roulette Wheel Selection: The pie is split among chromosomes of fitness 10, 80 and 90. Portions of the pie are allocated to them proportionally to the fitness. 36
Figure 8: Rank Selection: The pie is split among chromosomes of fitness 10, 80 and 90. Chromosomes are ranked from 1 to 3 and portions of the pie are allocated to them proportionally to the rank. 37
Figure 9: A general functional model describing the hybrid heuristics. . 50
Figure 10: The graph representation. 53
Figure 11: Parents before rule crossover (each chromosome is identified by a color). 55
Figure 12: The children after rule crossover. 56
Figure 13: Parents before condition crossover (Edges chosen are labeled with a *). 58
Figure 14: The children after condition crossover. 58
Figure 15: Chromosome before operator mutation on edge linking attribute node "Stress" to value node "0.6". 60
Figure 16: Chromosome after operator mutation on edge linking attribute node "Stress" to value node "0.6". 60
Figure 17: Chromosome before class mutation (the yellow node indicates the mutation point). 61

Figure 18: Chromosome after class mutation (The yellow node indicates the vertex that was affected by mutation). 62
Figure 19: Chromosome before value mutation occurring on the yellow vertex.. 63
Figure 20: Chromosome after value mutation occurring on yellow vertex (the value was changed from 5 to 4). 64
Figure 21: Chromosome before condition addition mutation. 65
Figure 22: Chromosome after adding the two nodes shown in yellow and the edge that connects them. 66
Figure 23: Bar Chart showing the results of the three hybrid heuristics compared to C4.5 in Experiment 1 done on STAB2..................... 83
Figure 24: Bar Chart showing the results of the three hybrid heuristics compared to C4.5 in Experiment 2 on STAB2............................ 85
Figure 25: Bar Chart showing the results of the three hybrid heuristics compared to C4.5 in Experiment 3 on STAB1............................ 87
Figure 26: Plot showing the improvement in accuracy in Experiment 1 on STAB2 at the three phases of optimization for each hybrid heuristic. 92
Figure 27: Plot showing the improvement in J-Index in Experiment 3 on STAB1 at the three phases of optimization for each hybrid heuristic. 93

List of Tables

Table 1: An example of classes with their corresponding complexity metrics.. 10
Table 2: STAB1 software quality metrics used as attributes in the classifiers (Azar, 2004). 73
Table 3: STAB1-Software systems used to build classifiers with C4.5 (Azar, 2004). 74
Table 4: STAB1- Software systems used to train and test the heuristics (Azar, 2004). 74
Table 5: STAB2-Software quality metrics used as attributes in the classifiers (Azar, 2004). 76
Table 6: STAB2-Software systems used to build classifiers with C4.5 (Azar, 2004). 77
Table 7: STAB2-Software systems used to train and test the heuristics (Azar, 2004). 77
Table 8: Settings and parameters for the SA-GA hybrid heuristic......... 79
Table 9: Settings and parameters for the TS-GA hybrid heuristic. 80
Table 10: Settings and parameters for SA-TS-GA hybrid heuristic....... 81
Table 11: Results of the three hybrid heuristics compared to C4.5 in Experiment 1 done on STAB2. Standard deviation is shown in parenthesis.. 82
Table 12: Results of the three hybrid heuristics compared to C4.5 in Experiment 2 on STAB2. Standard deviation is shown in parenthesis.. 84
Table 13: Results of the three hybrid heuristics compared to C4.5 in Experiment 3 on STAB1. Standard deviation is shown in parenthesis.. 86
Table 14: Table showing a comparison of SA-GA results with previous results on STAB2. Standard deviation is shown in parenthesis... 89

Table 15: Table showing a comparison of SA-GA results with previous results on STAB1. Standard deviation is shown in parenthesis... 89
Table 16: Size of rule sets on STAB1. 90
Table 17: Size of rule sets on STAB2. 90

CHAPTER 1: INTRODUCTION

Nowadays, virtually all societies rely heavily on complex computer-based systems. The increasing dependability on software systems has intensified the demand for and pursuit of high quality software. As a consequence, software quality assurance and prediction have become a crucial complement to the software development process.

Estimating software quality in the early stages of software development helps reduce cost, effort and time (Galin, 2004). Assessment of software quality is a grueling task since most software quality characteristics such as maintainability, reliability and reusability cannot be measured before the software product is deployed and in use. However, these characteristics can be predicted from other software quality attributes such as complexity and inheritance (Abreu & Melo, 1996), (Azar, 2004), (De Almeida & Matwin, 1999), (Fenton & Neil, 1999). Some of these attributes can be measured during the early stages of software development by the evaluation of software metrics. Examples of such metrics are the Chidamber Kemerer (CK) metrics (Chidamber & Kemerer, 1994) that comprise, among others, NOC (Number of Children of a class in an object oriented software system) and DIT (Depth of Inheritance tree). These metrics are used to build software quality estimation models that take different forms: statistical models, rule based models and decision trees (De Almeida & Matwin, 1999), (Khoshgoftaar, Allen, Halstead, Trio & Flass, 1998). In our work, we will work with rule based models because they have a white box nature that renders them easy to interpret by human experts. Many software quality estimation models have been built and used since the early 1970's (Pan,

1999). However, since data used to build such estimation models is scarce in the software development field, representative samples cannot be obtained. As a result, the accuracy of these models deteriorates when they are applied to new unseen data.

In this thesis, we shall tackle this problem by developing a hybrid heuristic approach to optimize a set of existing rule based models by combining and adapting them to the new context specific datasets. Our goal is to transfer the knowledge already learned from the old estimation models into the new adapted models.

1.1 Defining Software Quality

Many definitions have been proposed for software quality. According to (Pressman, 1988), Software quality is defined as:

> "Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software."

It is clear from the above definition that software quality is a product of different quality characteristics.

1.1.1 Software Quality Characteristics

In this section, we define important software quality characteristics such as reliability, maintainability, etc.

***Reliability*:** The probability of failure-free operation over a specified time in a given environment for a specific purpose (Pan, 1999)

Usability: Usability requirements deal with the scope of staff resources needed to train a new employee and to operate the software system (Galin, 2004).

Maintainability: The ease with which a certain software component can be modified to correct failures or to add new functionalities (Galin, 2004).

Availability: The probability that a system, at a point of time, will be operational and able to deliver the requested services (Sommerville, 2004).

Efficiency: Efficiency requirements deal with the hardware resources needed to perform all the functions of the software system in conformance to all other requirements (Galin, 2004).

Stability: A class in an object-oriented system is said to stable if its public interface remains the same across consecutive versions of the system.

Most of these characteristics cannot be assessed before the software product's deployment and use. However they can be predicted from the evaluation of certain software quality attributes such as size and complexity. Many metrics have been proposed in order to measure these attributes. In the next section, we describe some software quality attributes and the metrics used to measure them.

1.1.2 Software Quality Attributes and Metrics

As previously mentioned, software quality characteristics can be estimated and predicted during the early stages of software development by the assessment of software quality attributes such as size and complexity. Below, we describe software quality attributes and give examples of metrics to measure them.

The Size Attribute

The size attribute is the most widely measured software quality attribute. Measuring the size of a software product before its release is essential especially if we want to predict the future maintainability of the product (Kan, 2002).

Examples of metrics used to measure code size include:

LOC (Lines of Code): Total Lines Of Code: All lines that are not comments or blanks (Andersson, 1990).

LOC per Class: The average LOC value per class in object-oriented systems (Kan, 2002).

LOC per Method: The average LOC value per method in object-oriented systems (Kan, 2002).

Total number of operators: This is the total number of executable verbs in a program (Andersson, 1990).

Total number of operands: This is the total number of data names and literals in a program (Andersson, 1990).

The Complexity Attribute

Complexity is another software quality attribute that affects the maintainability, testability and reusability of a software system (Andersson, 1990). Curtis defines complexity as "a characteristic of the software interface which influences the resources another system will expand or commit while interacting with the software" (Curtis, 1980).

In other words, complexity is usually determined by control variables such as if-then statements whose values usually determine the paths to be taken (Andersson, 1990).

Examples of metrics used to measure complexity in Object-Oriented software are the CK (Chidamber and Kemerer) metrics that were initially proposed in 1994 (Kan, 2002).

These metrics include:

Response for a Class (RFC): This is the number of methods that can be executed in response to a message received by an object of that class (Kan, 2002).

Coupling Between Object Classes (CBO): An object class is coupled to another one if it invokes one's member functions or instance variables (Kan, 2002).

Number of Children of a Class (NOC): This is the number of immediate successors (subclasses) of a class in the hierarchy (Kan, 2002).

In the subsequent section, we explain how such metrics are used to build software quality estimation models.

1.2 Software Quality Estimation Models

Software quality estimation models are built from software quality metrics. Their goal is to estimate or predict immeasurable software quality characteristics such as reliability based on measurable attributes such as size and complexity. Software quality estimation models are usually classified into two main categories: statistical models and logical models. Examples of logical models include rule sets and decision trees (Jones & Khoshgoftaar, 1998), (De Almeida & Matwin, 1999). Examples of statistical models include regression models (Abreu & Melo, 1996), (Khoshgoftaar, Allen, Halstead, Trio, & Flass, 1998). In our work, we will deal with rule sets because they have a white box nature that makes them easy to interpret by human experts.

To illustrate, let us consider the task of predicting the maintainability of a class in object oriented systems based on the complexity attribute. Maintainability is defined as the ease with which a certain software component can be modified to correct failures or to add new functionalities (Galin, 2004). Let us also assume that the rule-based

model is built from the three complexity metrics previously defined: NOC (Number of Children of a Class), RFC (Response for a Class) and CBO (Coupling Between Object Classes). We shall denote by 0 the fact that a class is maintainable and by 1 the fact that it is not maintainable. The values 0 and 1 are called *classification labels*. An example of a rule set built from the three complexity metrics is shown as follows:

Rule 1: $CBO > 2 \ \& \ RFC \leq 1 \ \rightarrow 0$

Rule 2: $NOC > 3 \ \& \ CBO \leq 2 \ \rightarrow 1$

Default Class: 0

The rule based model consists of two rules and a default classification label. This model is read as follows: Rule 1 states that if a class has a CBO greater than 2 and RFC less than or equal to 1 then this class is maintainable (classification label 0). Rule 2 states that if a class has a NOC greater than 3 and a CBO less than or equal to 2 then the class is not maintainable (classification label 1). Finally, the default class indicates that if a class does not satisfy the conditions of Rule 1 and Rule 2, then it will be classified as maintainable (classification label 0).

This model can be used to predict the maintainability of classes based on the complexity metrics. To illustrate the classification process, let us suppose we want to predict the maintainability of the classes in the dataset shown in Table 1. The dataset describes each class in terms of its corresponding metric' values.

	CBO	**RFC**	**NOC**
Class 1	3	1	4
Class 2	1	3	5
Class 3	4	2	2

Table 1: An example of classes with their corresponding complexity metrics.

The model starts with the first class (Class 1). Then it applies the first rule and checks if the class satisfies the rule's conditions. In the case of class 1, CBO is 3 so the condition CBO>2 is satisfied. Since RFC is equal to 1 then the condition RFC≤1 is also satisfied. Hence, since all conditions of Rule 1 are satisfied, the models attributes to Class 1 the classification label 0 of Rule 1 indicating that the class is maintainable. Class 2 does not satisfy the conditions of Rule 1 but satisfies the conditions of Rule 2 so it will be classified by Rule 2 as not maintainable (classification label 1). Class 3 is not classified by neither rule. Hence Class 3 will be given the default class of 0 (maintainable).

1.3 Problem Statement and Objective

As we have seen, rule sets are easy to interpret by human experts due to their white-box nature. However, since data used to build such rule sets is insufficient in the software quality field, representative samples cannot be obtained. As a result, the accuracy (percentage of correctly classified cases) of these models deteriorates when they are applied to new unseen data. The core aim of this thesis is to deal with the optimization of already existing rule based models by adapting them to new unseen data.

In (Azar, 2004), Azar presents two genetic algorithm based approaches to adapt already existing models to new unseen data. In the first approach, Azar proposes the adaptation of a single model, in the second one Azar proposes the combination of several models. Both approaches proved to improve the models. In the future work section, the author suggests the use of a hybrid heuristic approach to improve software quality estimation models. In this thesis, we plan to investigate such hybrid approach by implementing and testing three hybrid heuristics

and comparing their results with the results obtained in (Bouktif, Azar, Sahraoui, K´egl & Precup, 2004) and (Azar 2004).

More precisely, the hybrid heuristics will combine and adapt the already existing rule sets to the new context specific datasets and produce final rule sets that are more accurate than the initial ones.

Our approach will be hybrid in both structure and nature: On one hand, its algorithmic design will merge the skills of different heuristic techniques such as Genetic Algorithms, Tabu Search and Simulated Annealing. On the other hand, it will simultaneously work on two levels of optimizations:

- Optimization of a single model by adapting it to a dataset.
- Optimization of a set of models by combining and adapting them to a dataset.

1.4 Thesis Organization

The rest of this thesis is ordered as follows: Chapter 2 offers an informational background on inductive learning as well as an overview of previous and related work in building software quality estimation models. Chapter 3 gives a detailed explanation of genetic algorithms. Chapters 4 and 5 present respectively the Simulated Annealing and Tabu Search heuristics. Chapter 6 thoroughly depicts the design and algorithmic configuration of our three hybrid heuristics as well as the experimental results assessing the performance of the heuristics. Finally, Chapter 7 concludes the thesis and proposes some future exploration paths in the domain of software quality estimation models' optimization.

CHAPTER 2: BACKGROUND

In this chapter, we present an informational background on inductive learning and shed light on previous and related work in building software quality estimation models.

More precisely, in the first part of the chapter, we explain the evaluation criteria used to assess software quality estimation models. We also offer an overview on logical classification models, namely decision trees and rule sets. Since we use rule sets that are built using *C4.5*, we also explain how this algorithm builds decision trees and rule sets.

In the second part of this chapter, we show previous work in building statistical and logical software quality estimation models as well as related work in optimizing already existing rule-based software quality estimation models.

2.1 Inductive Learning Background.

Inductive learning, also called concept learning, consists of "inferring a boolean-valued function from training examples of its input and output" (Mitchell, 1997). In an informal way, inductive learning consists of acquiring *general concepts* from *specific training examples.* Training examples are typically structured as a set of attributes along with a classification label. The Boolean-valued function predicts the classification label based on the set of values for the attributes. Inductive learning relies on a major assumption called the inductive learning hypothesis stating that: "Any hypothesis found to approximate the target

function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples" (Mitchell, 1997).

To illustrate, let us consider the example of estimating the maintainability of a class. Table 2 shows a potential set of training examples. Each row describes a case which represents a class in an object-oriented software system. The attributes in our case represent complexity metrics: NOC (Number of Children of a Class) , RFC (Response for a Class) and CBO (Coupling Between Object Classes). The classification label is the software characteristic we wish to predict: Whether or not the class is maintainable. Each column except the last one is a value that the class has for the corresponding metric and the last column indicates the classification label (maintainable or not).

RFC	NOC	CB	Maintainable
3	2	4	**No**
4	3	15	**Yes**
3	3	5	**No**
5	6	10	**Yes**

Table 2: An example of training instances.

From these training instances, a classification model can be built using inductive learning. In the next section, we take a deeper look at classification models, namely decision trees and rule sets and we explain the *C4.5* algorithm, a rule set and decision tree builder.

2.1.1 Classification Models

Software estimation models are usually cast into two main categories: statistical and logical models. The major drawback to using statistical models is their black box nature whereas they fail to show any true causal relationship explaining how the prediction is attained (Fenton & Neil, 1999). Consequently, in this thesis, we shall only deal with logical models, specifically rule sets, because they have a white box nature making them easy to interpret by human experts.

In the next sections, we explain the *C4.5* algorithm, a decision tree and rule set builder, that was used to build the initial rule based estimation models to be optimized by the hybrid heuristics. Furthermore, we take a deeper look at two logical classification models: Decision Trees and Rule Sets.

2.1.1.1 C4.5 Algorithm

C4.5 is a decision tree algorithm that was developed by Dr. Ross Quinlan in 1993. The basic algorithm to build a tree works as follows (Quinlan, 1993):

Given a set of training instances:

1: if all the training instances have the same class label, create a leaf with this class label and exit.

2: Pick the best test that splits the data

3: Split the training set according to the value of the outcome of the test.

4: Recurse on each subset of the training data.

The decision tree built can then be easily converted to a rule set.

To illustrate let us consider the training instances used to predict stability in Table 3.

NOC (Number of Children)	DIT (Depth of Inheritance)	NOM (Number of Methods)	**Stable**
3	2	4	**No**
4	3	7	**No**
3	3	5	**No**
5	6	10	**Yes**

Table 3: An example of training instances.

From these training instances, C4.5 builds a decision tree which can then be converted to a rule set. This rule set can then be used to predict classification labels for new instances (these new unseen instances constitute the testing set).

Figure 1 shows an example of the decision tree and the corresponding rule set.

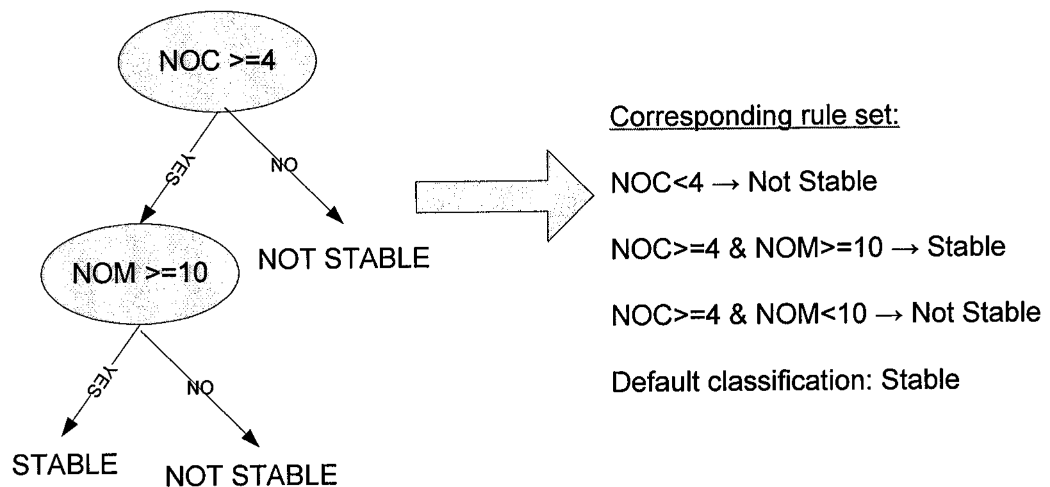


Figure 1: A decision tree and its corresponding rule set.

Appropriate problems where we can apply *C4.5*, or, in general decision tree and rule based learning are problems that typically have the following characteristics:

- Instances are represented by attribute-value pairs: For example, in our case, an instance represents a class with a set of attributes (software metrics) and their respective values.
- The target function has discrete values: For example, Yes or No to determine maintainability.
- The training data may contain errors.
- The training data may contain missing attribute values.

A major issue that can be encountered when building decision trees or rule sets is called *overfitting.* Overfitting occurs when the classification model performs well on the training dataset but poorly when applied to new unseen data forming the testing dataset. To avoid data overfitting, C4.5 performs *pruning* on the decision tree by greedily removing the nodes which removal most improves the accuracy on the testing dataset (Qunilan, 1993).

In the next two sections, we give a thorough overview of the two classification models built by *C4.5:* Decision Trees and Rule Sets.

2.1.1.2 Decision Trees

Decision trees are logical classification models that have a tree-based structure. They classify instances by sorting them down the tree from the root to some *leaf node* which represents a classification label. Each *node* in the tree specifies a test of some attribute of the instance, and each branch descending from that node corresponds to one of the possible values for that attribute. The classification process is done as follows: Starting at the root node, we test the attribute specified by this node, then we move down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the subtree rooted at the new node until we reach a leaf node that will determine the classification label of the example (Mitchell, 1997).

To illustrate, let us consider the decision tree shown in Figure 2 predicting the maintainability of a class based on three complexity metrics: RFC, CBO and NOC. Let us consider we wish to classify the case of Class 1 with the following attributes:

Class1: RFC=3; NOC=2; CBO=1

Looking at Figure 2, we start with the root node and perform the test for NOC. In our case NOC is equal to 2 which is less than 3 so we branch down to the left. Now we perform the test for RFC. In our case RFC is equal to 3 which is greater than 2 and we consequently follow the branch to the right. Now the iteration stops since we have reached a leaf node. The decision tree classifies the case as YES (maintainable).

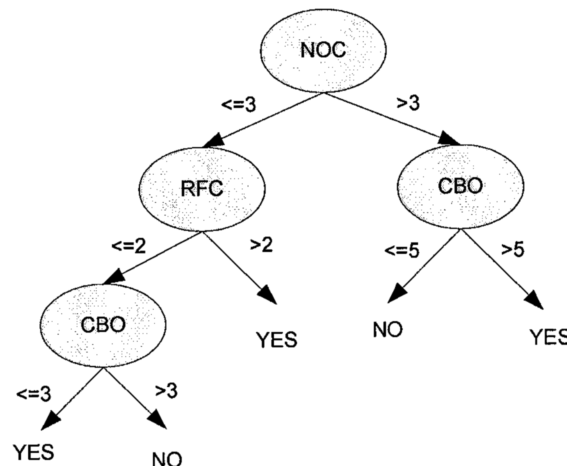


Figure 2: A decision tree predicting the maintainability of class based on complexity metrics.

2.1.1.3 Rule Sets

Rule-based estimation models are amid the most expressive and human readable classification models. In fact, due to their white-box nature, they have become quite popular in the software quality estimation field. A rule set is composed of a set of rules and a default

classification label. Each rule is composed of a conjunction of conditions and a classification label.

Classification of a case is done sequentially: A case will be given the classification label of the first rule that classifies it. If no rule classifies the case, then the latter will be given the default classification label of the rule set.

To illustrate, let us consider the following rule set predicting the maintainability of a class based on complexity attributes: RFC, NOC and CBO. A classification label of 0 indicates that the class is maintainable and a classification label of 1 indicates that the class is not maintainable.

Rule 1:	CBO > 2 & RFC ≤ 1		→ 0
Rule 2:	NOC > 3	& CBO ≤ 2	→ 1
Rule 3:	NOC > 3	& CBO ≤ 5	→ 0
Rule 4:	NOC > 3	& RFC ≤ 2	→ 1
Rule 5:	RFC> 3	& CBO ≤ 2	→ 0
Default Class:	0		

Let us suppose we want to classify the following case:

Class 1: NOC=4; CBO=1, RFC =4.

Rule 2 is the first rule that would classify the case and the classification label 1 would be attributed. The class would be classified as not maintainable.

In the next section, we define evaluation criteria we use to assess the "goodness" of the classification models, namely the rule sets.

2.1.2 Evaluation Criteria

The evaluation criteria determine the quality of a classification model by measuring a percentage of correctly classified cases. The

following are some definitions of the main notations used in evaluation criteria.

The real classification label:

This is the classification label that has been recorded during data collection (Quinlan, 1993).

The predicted classification label:

This is the classification as given by the classification model to the case (Quinlan, 1993).

The confusion matrix:

This is a k*k table, where k is the number of classification labels, the computed on a dataset D where each entry n_{ij} in the table indicates the number of cases in D with a real classification label i and predicted classification label j (Quinlan, 1993).

Predicted label

Real label	c_1	c_2	...	c_k
c_1	n_{11}	n_{12}	...	n_{1k}
c_2	n_{21}	n_{22}	...	n_{2k}
⋮	⋮	⋮	⋮	⋮
c_k	n_{k1}	n_{k1}	...	n_{kk}

Table 3: The confusion matrix.

The diagonal represents the number of cases that are correctly classified since their real label and predicted label are the same.

The accuracy:

The accuracy of a classification model M computed on a data set D is the percentage of cases in D correctly classified by M:

$$C(M) = \frac{\sum_{i=1}^{k} n_{ii}}{\sum_{i=1}^{k} \sum_{j=1}^{k} n_{ij}}$$

The J-Index:

The J-Index of a classification model M is the average accuracy per class label and is given by the following formula:

$$J(M) = \frac{1}{k} \sum_{i=1}^{k} \frac{n_{ii}}{\sum_{j=1}^{k} n_{ij}}$$

In our work, we shall work on the improvement of both the accuracy and the J-Index of the rule-based software quality estimation models. We use the J-Index as well because one of the datasets that we use in our experiments is unbalanced. On such a dataset, the majority classifier will always have a high accuracy although it does not classify correctly any of the minority cases. Hence, it would be more important to give more weight to the minority cases. For this, the J-Index is used to see on average how good the model is on each classification label.

2.2 Previous and Related Work

In this part of the chapter, we introduce previous work in building software quality estimation models as well as related work in optimizing already existing models.

2.2.1 Previous Work in Building Software Quality Estimation Models

In this section, we shed light on previous work in building statistical and logical models.

2.2.1.1 Previous Work in Building Statistical Software Quality Estimation Models

Previous work in building statistical software quality estimation models includes the works mentioned in (Khoshgoftaar, Allen, Halstead, Trio & Flass, 1998), (Gao & Khoshgoftaar, 2007) and (Abreu & Melo, 1996).

In (Khoshgoftaar, Allen, Halstead, Trio & Flass, 1998), a statistical model based on logistic regression is built in order to predict the reliability and fault-proneness of a software system. The model built is tested on a real-time military system called JSTAR (Joint Surveillance Target Attack Radar System). The objective of the model was to predict at the start of integration whether a module would be fault-prone or not fault-prone at the end of integration based on process metrics. The results showed that the models had an average misclassification rate of 35.5% for modules actually not fault-prone (Type I error) and an average misclassification rate of 21.5% for modules actually fault-prone (Type II error) and 31.8% overall, yielding an overall accuracy of 69.2%.

In (Gao & Khoshgoftaar, 2007), two statistical models, the Poisson regression model (PRM) and the Zero-Inflated Poisson (ZIP) regression model were built in order to predict the fault-proneness of a software

module based on fault density and fault count metrics. The models were tested on two large datasets: The first dataset, consisting of 282 program modules, was collected from a Command and Control Communication System (CCCS), which is a large military telecommunications system written in Ada using the procedural paradigm. The second dataset was taken from two large Window-based embedded applications used for configuration of wireless telecommunications products and comprised 1211 modules. The results of the two models were then compared with another Linear Regression Model (LRM). The results showed that the PRM model had an average accuracy of 83%, the ZIP model had an average accuracy of 84% while the LRM had an average accuracy of 83%.

In (Abreu & Melo, 1996), a study to assess the impact of Object-Oriented design on Software Quality is presented. Linear Regression statistical models are built to predict *defect density* (a ***reliability*** measure) and *normalized rework* (corrective maintenance effort, a ***maintainability*** measure) in object-oriented systems. The models are built based on data collected from metrics measuring software quality attributes such as polymorphism and inheritance. The results achieved allowed to confirm with an average confidence of 99% that design alternatives have a strong influence on resulting quality.

2.2.1.2 Previous Work in Building Logical Software Quality Estimation Models

The main drawback of statistical models is that they have a black box nature rendering them hard to interpret by human experts. Moreover, (Fenton & Neil, 1999) argue that statistical and regression-based models are usually based on limited historical data and fail to incorporate true causal relationships. Logical models have gained larger popularity in the software engineering field.

In (Fenton & Neil, 1999), a machine learning approach based on Bayesian Beliefs Networks (BBNs) was used to build software quality

estimation models that predict reliability (defect density) based on process and product metrics such as testing effort, design effort and problem complexity. Their ability to provide guidelines on how to attain the prediction makes them a powerful risk management decision support tool for managers. In fact, the BBNs offer predictions for variables such as testing effort, design effort in order to attain the defect density goal value (a reliability measure). Moreover, recent areas of application for BBNs include their celebrated use by Microsoft in the help wizards of Microsoft Office. They have also been used to predict general software quality attributes such as defect-density and cost in the IMPRESS project (Fenton & Neil, 1999).

In (Evett, Chien, Khoshgoftar & Allen, 1998), another machine learning technique based on Genetic Programming (GP) is used to build logical software quality estimation models that predict the relative fault-proneness of a module by assigning an ordinal rank to each module, thus classifying modules from most to least fault-prone. The models were tested on two large real datasets, one in Ada forming a total of 282 modules and the other in Pascal consisting of a total of 171 modules, and compared to random ordering techniques. The results showed that the top 10% of the modules ranked by the GP-based models accounted for 152 faults, yielding an accuracy of 63% of the total number of faults while the top 10% of random ranking accounted for only 23 faults, representing an accuracy of only 9%.

(So, Cha & Kwon, 2001), (De Almeida & Matwin, 1999) and (Jones & Khoshgoftaar, 1998) also used inductive learning techniques such as rule sets and decision trees to predict software quality characteristics. The characteristics that they were interested were reliability, maintainability and fault-proneness.

In (So, Cha & Kwon, 2001), a fuzzy logic rule-based approach to predict error-prone modules using inspection data based on the

evaluation of metrics such as "LOC inspected per hour" is developed. In this approach, the authors derive inference rules from fuzzy noisy data in order to predict the number of defects and errors per KLOC (thousand Line Of Code). Tested on 38 modules, the model accurately classifies 28 yielding an overall accuracy of 74%.

In (De Almeida & Matwin, 1999), the results of the application of five decision tree and rule set builder algorithms on real Cobol modules of the telephone company Bell Canada are presented in a comparative analysis. In fact, five different software quality estimation models were generated using the decision tree constructors NEWID (Boswell, 1990), CN2 (Clark & Niblett, 1989) , C4.5 Tree (Quinlan, 1993) and the rule set constructors C4.5 rules (Quinlan, 1993) and FOIL (Quinlan, 1990). These models were tested on the Bell Canada modules to predict alteration cost (a maintainability measure) based on code size and complexity metrics. The results showed that the overall accuracies of NEWID, CN2, C4.5 Tree and C4.5 rules did not differ significantly: In fact, C4.5 Tree, C4.5 rules and NewID had an average accuracy of 77% and CN2 had an average accuracy of 75%. FOIL's performance was slightly poorer with an average accuracy of 64.5%. The authors also concluded with a note stating that C4.5 rule sets were the "most comprehensible", a statement supporting the hypothesis we previously mentioned about rule-based software quality estimation models being preferred by human experts.

Another study where decision trees were used to build software quality estimation models is shown in (Jones & Khoshgoftaar, 1998). In this paper, decision tree software quality estimation models are built using CART to predict the fault-proneness of software modules. The experiments conducted were based on real datasets taken from large telecommunication systems. The results showed an overall accuracy of 72.6% of the models.

2.2.2 Related Work in Optimizing Existing Rule Based Software Quality Estimation Models

The central concern of this thesis is the optimization of already existing rule-based software quality estimation models. The primary related work is found in (Azar, 2004) and (Bouktif, Azar, Sahraoui, K´egl & Precup, 2004) where two genetic algorithm based approaches are presented and compared. The first combines and adapts a set of rule sets to a dataset. The second adapts a single model (rule set) to a dataset. The initial models are generated by *C4.5*. In (Bouktif, Azar, Sahraoui, K´egl & Precup, 2004), the data experimented with is taken from the stability of Object Oriented classes. In (Azar, 2004), the genetic algorithms are tested on additional datasets describing the maintainability of Object Oriented classes. The experimental results show the improvement in both accuracy and J-Index of the models generated by the genetic algorithms over the initial *C4.5* models. Furthermore, the approach optimizing a set of models yielded superior improvement when compared to the one optimizing a single model.

Moreover, in (Azar, Precup, Bouktif, Kegl & Sahraoui, 2002), another machine learning technique called AdaBoost is used to optimize already existing rule-based software quality estimation models. The comparative results of AdaBoost and the genetic algorithms show that the genetic algorithm-based approaches outperformed AdaBoost.

In (Azar, 2004), the use of hybrid heuristic approaches that would combine the skills of genetic algorithms with other heuristics is proposed as a future work path in the optimization of already existing software quality estimation models. The goal of our work is to investigate the use of such hybrid heuristics by designing, implementing and testing three hybrid heuristics and comparing their results with the results obtained in (Bouktif, Azar, Sahraoui, K´egl & Precup, 2004) and (Azar 2004).

Our approach will be hybrid in algorithmic structure since it will be inspired from three different optimization techniques: Genetic Algorithms, Tabu Search and Simulated Annealing. Moreover, it will simultaneously work on the two levels of optimizations mentioned in (Azar, 2004) and (Bouktif, Azar, Sahraoui, K´egl & Precup, 2004): Optimization of a single model and optimization of a set of models.

CHAPTER 3: GENETIC ALGORITHMS

In this thesis, we deal with the optimization of rule sets using hybrid heuristics. The search space in this problem is large. Hence, genetic algorithms are chosen as a major component of the hybrid heuristics for their power to learn from a very large search space.

Genetic Algorithms (GAs) were first introduced by Holland in the 1970's (Holland, 1975). Inspired by the Darwinian theory of evolution (Darwin, 1859), they provide a learning method motivated by an analogy to biological evolution.

In the next section we give a brief overview of the Darwinian Theory behind Genetic Algorithms and define some essential biological terms that are used when dealing with Genetic Algorithms.

3.1 The Darwinian Theory

According to the Darwinian theory of evolution, individuals exist in an environment and compete for survival. The "fittest" individuals with the most favorable "traits" have a higher chance of survival and progeny production (Darwin, 1859). In a GA, the individuals existing in the environment and that undergo reproduction are called *chromosomes.* Typically, a chromosome can be seen as a string of basic units, each representing a particular trait. These basic units are called *genes,* and the set of values that the genes can have are called *alleles.* The position of a gene in a string is called its *locus* (Holland, 1975). The *genotype* represents the actual genetic structure representing the individual. The

phenotype represents the observed characteristics of the individual (Reeves & Rowe, 2003).

In a GA, typically each *chromosome* represents a solution to the optimization problem. The way the solution is represented in the chromosome is called the *encoding scheme*. In our case, each chromosome will represent a rule set. The *fitness* of the chromosome should reflect the goodness of the solution it represents. The process of *evolution* involves many iterations during which new chromosomes are created from existing ones. Each iteration is called a *generation*. The set of chromosomes present at a generation is called a *population*. Chromosomes that undergo reproduction are selected according to their *fitness* which correlates to the goodness of the solution. These selected chromosomes go through some genetic operations that lead to the creation of a new generation of child chromosomes. The most common genetic operators are *crossover* and *mutation*. Crossover is an operation whereby a pair of chromosomes exchange some of their genes to form new chromosomes (Reeves & Rowe, 2003). Mutation consists of changing allele values for one or more chosen genes of the chromosomes (Reeves & Rowe, 2003). Since chromosomes are selected according to their fitness, it is hoped that children chromosomes will gather dispersed "good traits" and thus have a higher fitness than their parents.

3.2 The algorithm

The general pseudo-code of a genetic algorithm is described as follows (Reeves & Rowe, 2003):

- Choose an initial population of chromosomes
- **While** termination condition not satisfied **do**
    - **Repeat**
        - **If** crossover condition satisfied **then:**
            - Select parent chromosomes
            - Perform crossover with a certain probability
        - **If** mutation condition satisfied **then:**
            - Select chromosomes for mutation
            - Choose mutation points in offspring
            - Perform mutation
        - Evaluate fitness of offspring
    - **Until** sufficient offspring created
    - Replace old population with new one
- **End While**

Before we take a deeper look at the genetic operators, specifically mutation and crossover, we illustrate common encoding schemes usually used to represent chromosomes.

3.2.1 Encoding Schemes

The encoding scheme refers to the way a solution is represented as a chromosome in a GA. The most common encoding schemes are the *binary representation* and the *integer/real-valued representation.*

In a binary representation, a chromosome is represented as a string of 0's and 1's. This representation is highly popular as it makes genetic operators such as crossover and mutation easy and straightforward to implement (Mitchell, 1997). This representation is ideal for problems whose solutions can be represented as a set of boolean-valued variables. To illustrate, let us consider the task of predicting whether a student will *pass* or *fail* a course based on two variables: *Analytical Skills* and *Writing Skills.* The chromosome that will represent a solution will be constituted of three genes: The first gene will represent the *Analytical Skills* variable; the second gene will represent the *Writing Skills* variables and the third gene will represent the prediction: *Pass/Fail.* In order to represent the alleles, that is, the set of values that each gene can have, let us suppose that the two variables *Analytical Skills* and *Writing Skills* can only take the four following discrete values: "Bad", "Good", "Very Good" and "Excellent". As a binary representation, we can assign the string 00 for allele "Bad", 01 for allele "Good", 10 for allele "Very Good" and 11 for allele "Excellent". For the prediction, we can assign 1 for allele "Pass" and 0 for allele "Fail". In order to represent a solution in a chromosome, we can concatenate the bit values for the two variables along with the bit value representing the prediction in a predetermined order determined by the *locus* that indicates the position of a gene in a chromosome. For example, the chromosome "00011" can be explained as follows: The first two bits 00 represent the value of the gene encoding *Analytical Skills.* The next two bits 01 represent the allele value "Good" for gene *Writing Skills,* and the

last bit 1 represents the allele value "Pass" for the gene that denotes the prediction.

However, the binary representation cannot be easily applied to all optimization problems. For example, permutation problems require an integer representation for the chromosomes where each integer would represent an element in the permutation problem. To illustrate, let us consider solving the Traveling Salesman problem. This problem can be defined as follows: Let n be the number of cities and $D=[d_{ij}]$ be the distance matrix whose elements d_{ij} denote the distance between city i and city j. The goal consists of finding the shortest tour visiting all cities exactly once (Cormen, Leiserson, Rivest & Stein, 2001). Now let us consider a TSP instance with 5 cities, represented by integers 1, 2, 3, 4 and 5. A solution to the TSP problem can be represented in a chromosome as a sequence of integers determining the tour. For example, the chromosome "14532" corresponds to the following tour: *"City 1 → City 4 → City5 → City 3→ City 2 → City 1"*. In this case, it would be hard to imagine a binary representation for a tour involving a large number of cities due to the fact that it would be very difficult to assign unique bit strings for each city.

3.2.2 Genetic Operators

The two main GA operators are crossover and mutation. Elitism is a third optional operator that we will use in our GA.

3.2.2.1 Crossover

The crossover operator yields two new offspring chromosomes from two parent chromosomes (Mitchell, 1997). The goal of crossover is to assemble dispersed traits from parent chromosomes to form fitter children chromosomes (Holland, 1975).

The most common forms of crossover are single-point crossover, double-point crossover and uniform crossover. To illustrate these crossover techniques, we will suppose a binary encoding scheme for the chromosomes.

3.2.2.1.1 Single-Point Crossover

In single-point crossover, the parent chromosomes are cut at a chosen position to form two parts. The first child chromosome receives the first part from the first parent and the second part from the second parent. The second child chromosome receives the first part from the second parent and the second part from the first parent (Reeves & Rowe, 2003).

Figure 3 shows an example of single-point crossover where parent chromosomes are cut after the fourth gene.

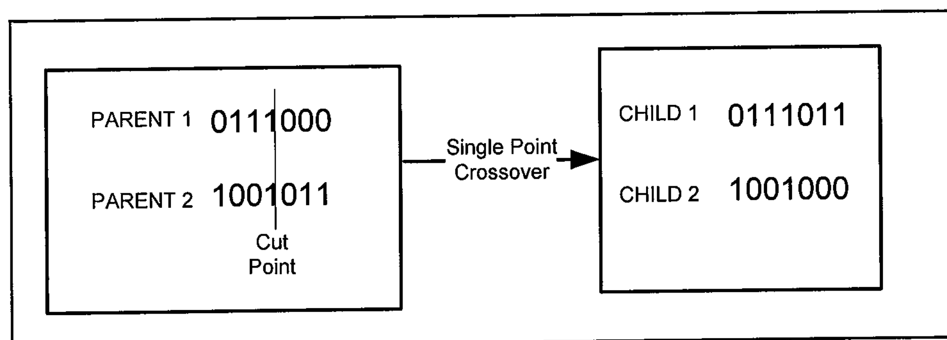


Figure 3: An example of single point crossover.

3.2.2.1.2 Double-Point Crossover

Double-point crossover is similar to single point crossover. However, we have two cut points at the parent chromosomes instead of one (Reeves & Rowe, 2003). The first child chromosome receives the first and third parts from the first parent and the second part from the second

parent. The second child chromosome receives the first and third parts from the second parent and the second part from the first parent.

Figure 4 illustrates double-point crossover where parent chromosomes are cut after the second and fifth genes.

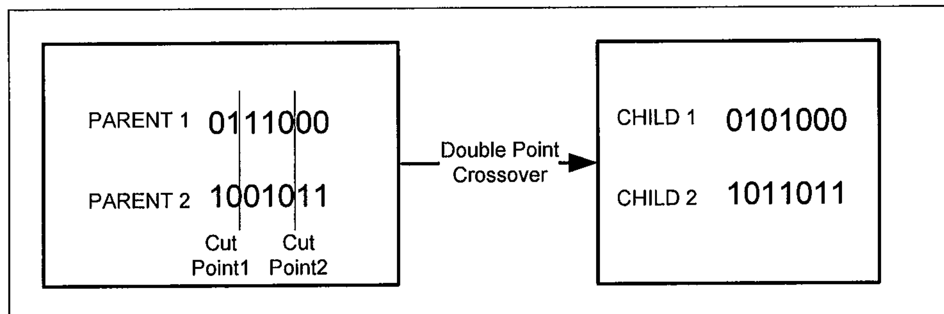


Figure 4: An illustration of double-point crossover.

3.2.2.1.3 Uniform Crossover

In uniform crossover, a random number of cut points is generated at random positions in the parent chromosomes (Mitchell, 1997). Child chromosomes inherit alternate parts from each parent.

Figure 4 shows an example of uniform crossover operation.

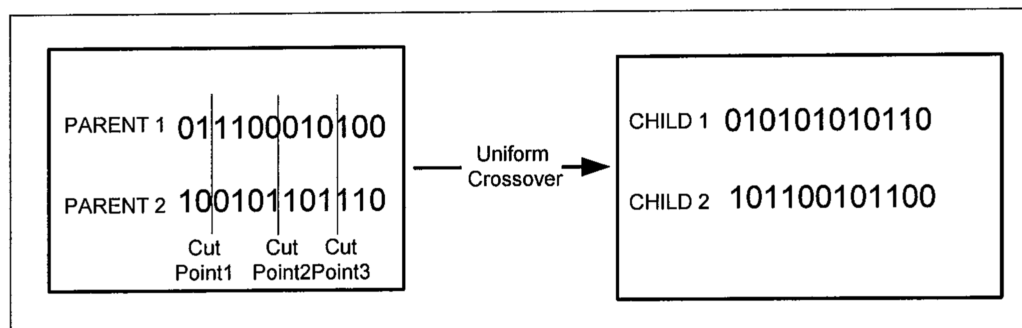


Figure 5: An illustration of uniform crossover.

3.2.2.2 Mutation

Unlike crossover that produces offspring by combining parts of two parents, mutation produces offspring from a single parent. Typically, mutation produces small random changes to the chromosome and

usually has a low probability of occurrence (Mitchell, 1997). Mutation usually occurs after crossover on the resulting offspring. In a binary representation, mutation can be seen as changing the value of a randomly chosen bit in the bit string representing the chromosome. Figure 6 shows a chromosome where the sixth gene is mutated and changed from 0 to 1.

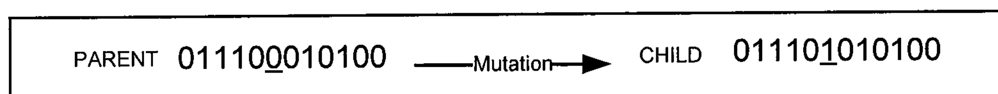


Figure 6: Single-bit mutation at position 6.

Many other mutation mechanisms have been proposed for integer or real-valued representations. They include value mutation and inversion mutation. In value mutation, the value of a chosen gene represented as a real value is altered. In inversion mutation, the order of a set of genes represented as integers is inverted.

3.2.2.3 Elitism

Elitism, introduced in (De Jong, 1975), is a process that aims to preserve the best solution(s) found during the evolution process. This is done by directly copying a percentage of the fittest chromosomes to the new generation.

3.2.3 Selection Schemes

Selection schemes define the basis on which chromosomes are selected to undergo genetic operators. The selection scheme is usually based on a measure of the chromosome fitness since, according to the Darwinian theory, individuals with the higher fitness should have a greater chance of survival and reproduction (Darwin, 1859).

In this section, we review the three most popular selection schemes: Roulette Wheel Selection, Rank Selection and Tournament Selection (Reeves and &, 2003).

3.2.3.1 Roulette Wheel Selection

In roulette wheel selection, the selection probability of an individual is directly proportional to the fitness of the individual. More precisely, the selection probability *S(i)* of individual *i* in a population of *k* individuals is given by the following equation:

$$S(i) = \frac{Fitness(i)}{\sum_{j=1}^{k} Fitness(j)}$$

To illustrate, the chart in Figure 7 represents the selection probabilities of three chromosomes: c_1 with fitness 90, c_2 with fitness 80 and c_3 with fitness 10.

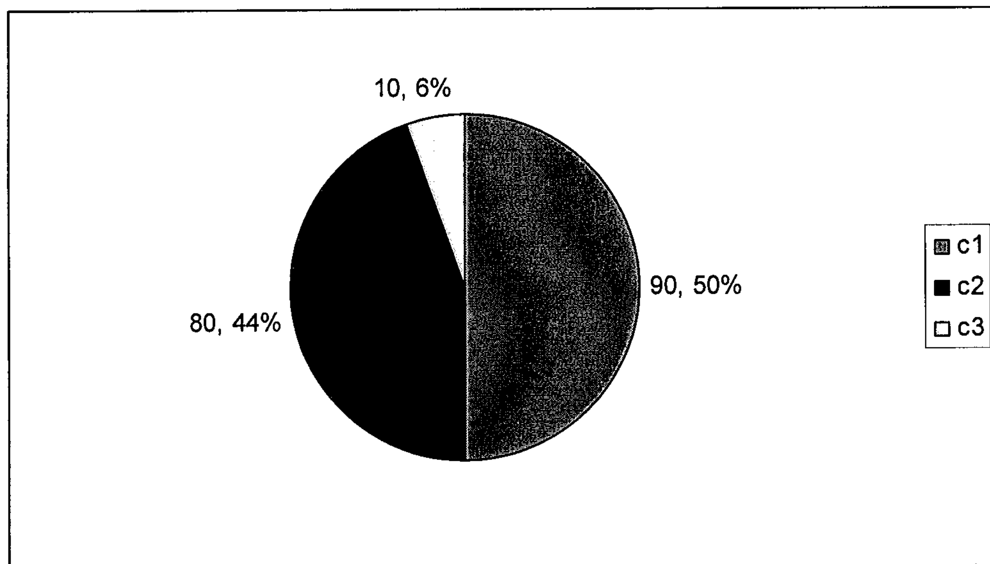


Figure 7: Roulette Wheel Selection: The pie is split among chromosomes of fitness 10, 80 and 90. Portions of the pie are allocated to them proportionally to the fitness.

3.2.3.2 Rank Selection

In rank selection, every chromosome is given a rank in accordance to its fitness. For instance, in a population of 3 chromosomes, the fittest one will be given rank 3, the next one rank 2 and the one with the lowest

fitness will be given rank 1. The selection probability of individual i in a population of k individuals is given by the following equation:

$$S(i) = \frac{Rank(i)}{\sum_{j=1}^{k} Rank(j)}$$

To illustrate, the pie in Figure 8 represents the selection probabilities of the three chromosomes shown in Figure 7.

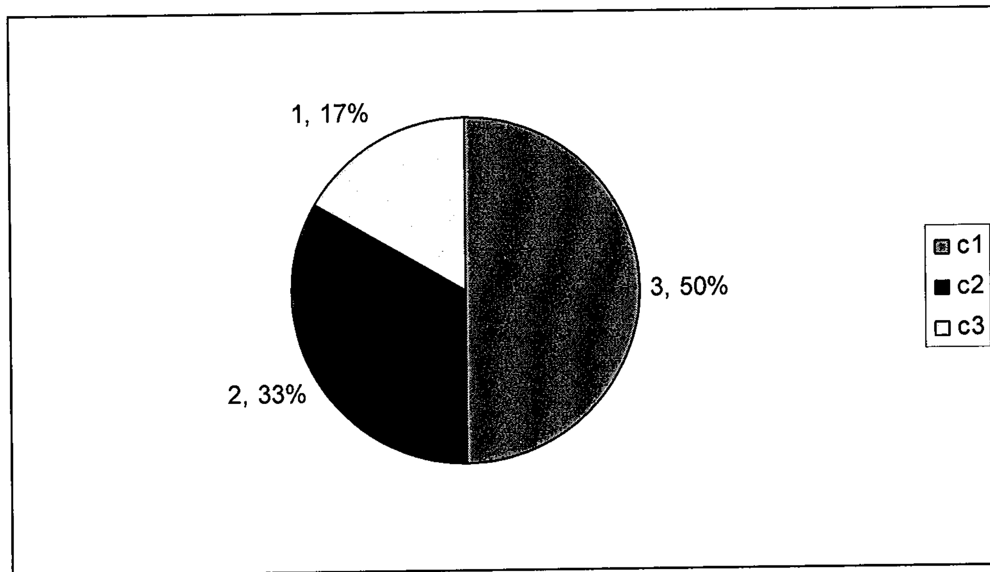


Figure 8: Rank Selection: The pie is split among chromosomes of fitness 10, 80 and 90. Chromosomes are ranked from 1 to 3 and portions of the pie are allocated to them proportionally to the rank.

Rank selection is preferred to roulette wheel selection if we want to give a higher chance of selection to individuals with very low fitness values. To illustrate, we can see that in roulette wheel, the selection probability of c3 is of 6% only while in rank selection, it becomes 17%.

3.2.3.3 Tournament Selection

Contrary to rank and roulette wheel selection schemes, tournament selection does not require a prior knowledge of the fitness values for all chromosomes in the population. In fact, in tournament selection scheme, k random chromosomes are selected from the current

population and the fittest of them is chosen as a first parent to undergo crossover. The remaining chromosomes are then returned to the population. The process is repeated for the selection of the second parent chromosome.

3.2.4 Replacement Policies

There are mainly two replacement policies for GAs: The first consists of replacing *all* chromosomes of the previous generation with the chromosomes of the new generation, generated by crossover, mutation or elitism. The GA is then called *generational GA* (Reeves & Rowe, 2003).The second policy consists of replacing only a fraction of the previous generation chromosomes with the new offspring. In this case, the GA is called a *steady state GA* (Reeves & Rowe, 2003).

3.2.5 Termination Conditions

GAs are search methods that could in principle run forever (Reeves & Rowe, 2003). Hence, a termination criterion is needed to stop the evolution process of new generation creation. Common termination conditions include:

- the attainment of a certain number of generations
- the attainment of a certain computer clock time
- the attainment of a limit or goal fitness value.

CHAPTER 4: SIMULATED ANNEALING

The second heuristic used in our hybrid techniques is Simulated Annealing. In this chapter, we present a thorough overview of the simulated annealing algorithm.

4.1 The Metropolis Algorithm: Boltzmann sampling and Monte Carlo Simulation.

4.1.1 Annealing: a thermal physical process.

Simulated Annealing (SA) is inspired from a physical thermal procedure used to make the strongest possible glass. The annealing process consists of the following two major steps (Barker & Henderson, 1976):

- Heating the solid to a maximum temperature at which it becomes liquid.
- Decreasing *slowly* the temperature of the solid until the particles arrange themselves in a most stable orientation. This slow and careful cooling is called *annealing.*

This physical procedure was first simulated in a simple algorithm proposed by Metropolis in the early 1950's (Metropolis, Rosenbluth, Rosenbluth, Teller & Teller, 1953). In the next section we give an overview of this algorithm.

4.1.2 The Metropolis Monte Carlo Algorithm

In (Metropolis, Rosenbluth, Rosenbluth, Teller & Teller, 1953), a simple algorithm known as the *Metropolis Monte Carlo algorithm* is introduced for simulating the evolution of a solid in a heat bath to *thermal equilibrium.*

The main goal of the algorithm is to minimize the energy E of the solid in thermodynamics and is based on the following method:

- Given a solid with a current state i and with energy E_i:
  - Apply a perturbation mechanism (e.g. displacement of a particle) to the solid.
  - Following the perturbation, the solid is now in a next state j. The energy of the next state is E_j.
  - If the *energy difference* $E_j - E_i$ is less than or equal to 0, the state j is accepted as the current state
  - If the energy difference is greater than 0, the state j is accepted with a certain probability given by:

$$\exp\left(\frac{Ei - Ej}{k_B T}\right)$$

where T denotes the temperature of the heat bath and k_B a physical constant known as *Boltzmann constant.* This acceptance rule is known as the *Metropolis criterion.*

If the decrease in temperature is done sufficiently slowly, the solid can reach thermal equilibrium at each temperature. In the next section we define the notion of thermal equilibrium that is directly related to Boltzmann distribution.

4.1.3 Boltzmann distribution

Thermal equilibrium is characterized by the *Boltzmann distribution* (Aarts & Korst, 1989). When the solid is heated, the speed at which the particles are moving will vary considerably: from very slow particles (low energy) to very fast particles (high energy). Most of the particles however will be moving at a speed very close to the average. The Boltzmann distribution shows how the speeds (and hence the energies) of a mixture of moving particles varies at a particular temperature.

In fact, this distribution shows the probability of the solid being in state *i* with energy E_i at temperature *T* and is given by

$$P_T\{X = i\} = \frac{1}{Z(T)} \exp\left(\frac{-Ei}{k_B T}\right)$$

where X denotes the current state of the solid and *Z(T)* is called the *partition function* which is calculated according to the following summation over all possible states *j*:

$$Z(T) = \sum_j \exp\left(\frac{-Ej}{k_B T}\right)$$

In the next section, we describe the simulated annealing algorithm and the encoding scheme used for combinatorial optimization problems.

4.2 The Simulated Annealing Algorithm

In the early 1980's, (Kirkpatrick, Gelatt & Vecchi, 1983) introduced the pioneering idea of applying annealing to combinatorial optimization problems. The concepts are based on an analogy between the physical

process and the problem of solving large combinatorial optimization problems.

In Simulated Annealing, the Metropolis algorithm can be applied to generate a series of solutions for a combinatorial optimization problem. For this, we assume that solutions of the problem are equivalent to states of a physical system and the cost of the solution is equivalent to the energy of the system. Hence, in analogy to the Metropolis algorithm where the goal is to find the state with the lowest energy, in Simulated Annealing, the goal is to find the solution with the lowest cost.

In Simulated Annealing, the variable taking the role of the temperature is called the *control parameter*. The simulated annealing can now be presented as an iteration of Metropolis algorithms, evaluated at decreasing values of the control parameter (Aarts & Korst, 1989).

In a more formal way, the pseudo-code for a Simulated Annealing algorithm can be formulated as follows:

Let T be the variable representing the temperature or control parameter.

Let T_0 be the initial temperature value.

Let *X[ ]* be an initial set of solutions.

Let K be a fraction between 0 and 1 representing the Boltzmann constant.

Let S represent a solution and f(S) represent the cost of solution S. The goal will be to find the solution S with the minimum cost f(S).

- Initialize T= T_0;
- **Repeat** until convergence or for a certain number of iterations:
    - **Repeat** for a certain number of trials for each temperature:
        - Randomly **choose** a solution S from the set X[ ].
        - **Perturb** solution S to obtain new solution S'
        - **Compute** the difference in energy *DE*:
            - DE = f(S') – f(S)
        - **If** (DE≤0)
            - **Accept** solution S' and update set of solutions X[ ].
        - **Else**
            - **Generate** a random number r between 0 and 1.
            - **If** $(r < e^{(-DE)/T})$
              Accept solution S' and update set of solutions X[ ].
            - **Else**
            - **Reject** solution S'
    - T = K * T
- **Return** X[ ].

The main differences between this algorithm and the Metropolis algorithm are the following:

- The Simulated Annealing algorithm repeats the Metropolis algorithm for a certain number of iterations
- The Simulated Annealing algorithm can optimize a set of solutions instead of a single one as in the Metropolis algorithm.

CHAPTER 5: TABU SEARCH

Tabu Search is the third heuristic component used in two of our hybrid heuristic techniques. In this chapter we give a comprehensive introduction to this heuristic technique.

5.1 Background

Tabu Search is a heuristic method that was originally proposed by Glover in the mid 1980's (Glover, 1986). This heuristic has been proven to be very efficient in providing solutions very close to optimality to various combinatorial problems (Gendreau, 2002) such as TSP (Traveling Salesman Problem), job scheduling and graph coloring. Current applications of Tabu Search span diverse domains such as VLSI design, financial analysis, resource planning, pattern classification and recognition, biomedical analysis, waste management, telecommunication and scheduling (Glover & Laguna, 2002).

The distinctive feature of Tabu Search is the use of memories called *tabu lists* that would prevent cycling back to a previously visited solution. The word tabu (or taboo) originates from a language of Polynesia and denotes things that are "forbidden to profane use or contact because of what are held to be dangerous supernatural powers" (Mish, 1993). Tabu Search allows local search methods to overcome local optima by storing solutions previously visited in short term memory (Gendreau, 2002). Local search is an iterative search process that starts

from a feasible solution and improves it by moving to a better solution that differs slightly from the previous one. Contrary to tabu search, local search stops when it encounters a local optimum. The main problem with local search is that often the local optimum is a mediocre solution. Tabu search helps overcome this problem by allowing non-improving moves and thus continuing the search even when a local optimum is found. Moreover, in order to avoid cycling back to a previously visited local optimum, Tabu Search stores the history of previously visited solutions in a memory called the tabu list (Glover & Laguna, 2002). In the next section we take a deeper look at the Tabu Search algorithm.

5.2 The Tabu Search algorithm

Before going through the details of the algorithm, let us define the elements of Tabu Search:

5.2.1 Elements of Tabu Search

The search space: This is set of all possible solutions that can be visited during the search (Gendreau, 2002).

The neighborhood structure: The neighborhood structure defines the local transformations that can be applied to the current solution. Formally, the neighborhood N of a solution S is given by:

N(S) = {solutions obtained by applying a single local transformation to S}

The transformation: The transformation is the function that defines a move in the local search. It is applied to the current solution to yield a new neighbor solution, slightly different from the old one.

The tabu list: The tabu list is a short-term memory used to store previously visited solutions. The period of time for which a solution stays in the tabu list is governed by a specified replacement policy. Examples of replacement policies include the following:

- If the size of the list is fixed and solutions are stored in a FIFO (First In First Out) manner.
- Solutions are stored in the tabu list for a specified number of iterations or CPU clock cycles.
- Solutions are stored in the list for a random number of iterations.

5.2.2 The algorithm

The algorithm for Tabu Search can be defined as follows:

Suppose we wish to minimize a function $f(S)$ over the domain of all possible solutions. We define:

- S as the current solution.
- S^* as the best solution found
- f^* as the value of S^*
- $N(S)$ as the neighborhood function of S
- $N'(S)$ as the admissible subset of $N(S)$ i.e. no solution in $N'(S)$ is in the tabu list.
- T as the tabu list.

- Given an initial solution *S0*:
    - *S=S0; f*=f(S0); S*=S0; T=ø*
    - **do**
        - **select** *S* in *argmin[f(S')]* where *S' Є N'(S)*
        - **if** *f(S)<f**
            - *f* = f(S)*
            - *S*=S*
        - **Record** *S* in *T*.
    - **while** termination criteria not satisfied

Examples of Tabu Search termination criteria are the following

- Reaching a certain number of iterations.
- A specified computer clock time has elapsed.
- Finding a solution S* satisfying a certain threshold or goal value.
- When no improvement is made for S* during a specified number of iterations.

5.2.3 Intensification and diversification

Intensification and diversification are optional additional elements to Tabu Search. Intensification consists of exploring more thoroughly the portions of the search space that seem "more promising" than others (Gendreau, 2002). For example, the search can be restarted from the best currently known solution and intensified on the neighborhood of that particular solution. Diversification is used to avoid the search from being stuck in a restricted portion of the search space. In fact, diversification consists of guiding the search process to unvisited regions and to generate solutions that differ in significant ways from those seen before. This can be done by forcing the search to restart at unexplored areas of the search space (Glover & Laguna, 2002).

CHAPTER 6: THE HYBRID HEURISTICS

DESIGN AND EXPERIMENTAL RESULTS

In this chapter, we present the three hybrid heuristics that constitute the core contribution of this thesis. In the first part of this chapter, we explain the design and algorithmic structure of our three hybrid heuristics. In the second part, we describe the experiments conducted to test the heuristics as well as the results obtained.

6.1 The Hybrid Heuristics: Algorithmic Design

6.1.1 The general algorithm

The main goal of the hybrid heuristics is to combine and adapt already existing rule sets to new context specific datasets by producing final rule sets that are more accurate than the initial ones. Moreover, the three heuristics will be hybrid both in nature and in algorithmic structure. In fact, they will combine the strengths of different heuristic techniques such as Genetic Algorithms, Tabu Search and Simulated Annealing. Furthermore, they will simultaneously work on two levels of optimization:

- Optimization of a single rule set by adapting it to a dataset.
- Optimization of a set of rule sets by combining them and adapting them to a dataset.

More precisely, the first hybrid heuristic, called **"SA-GA"** is a combination of Simulated Annealing and Genetic Algorithms. The second hybrid heuristic, called **"TS-GA"** combines Tabu Search and Genetic Algorithms. The third and last hybrid heuristic, called **"SA-TS-GA",** merges the skills of Tabu Search, Simulated Annealing and Genetic Algorithms.

Each hybrid heuristic works in a *three phase* optimization scheme as shown in Figure 9: We see that every C4.5 rule set is first individually optimized by a first heuristic (Simulated Annealing or Tabu Search). This is the first optimization phase of the hybrid algorithms done on a single rule set level. Then, the rule sets produced by this first phase will undergo a second phase of optimization done by a second heuristic (Genetic Algorithm) on the set of rule sets altogether. Subsequently, the best rule set produced by the GA will undergo a third and last optimization done by the Tabu Search or Simulated Annealing to yield the final rule set. At the end, this final rule set will be retrieved and compared with the best initial c4.5 rule set to assess the performance of the hybrid heuristics. The main difference between the three heuristics resides in the choice of the heuristic(s) for the first and third optimization phases:

- In the *SA-GA* heuristic, Simulated Annealing is used to optimize the rule sets individually.
- In the *TS- GA*, Tabu Search is used to optimize the rule sets individually.
- For the *SA-TS-GA* heuristic, either Simulated Annealing or Tabu Search is chosen based on an equal probability to optimize the rule sets individually.

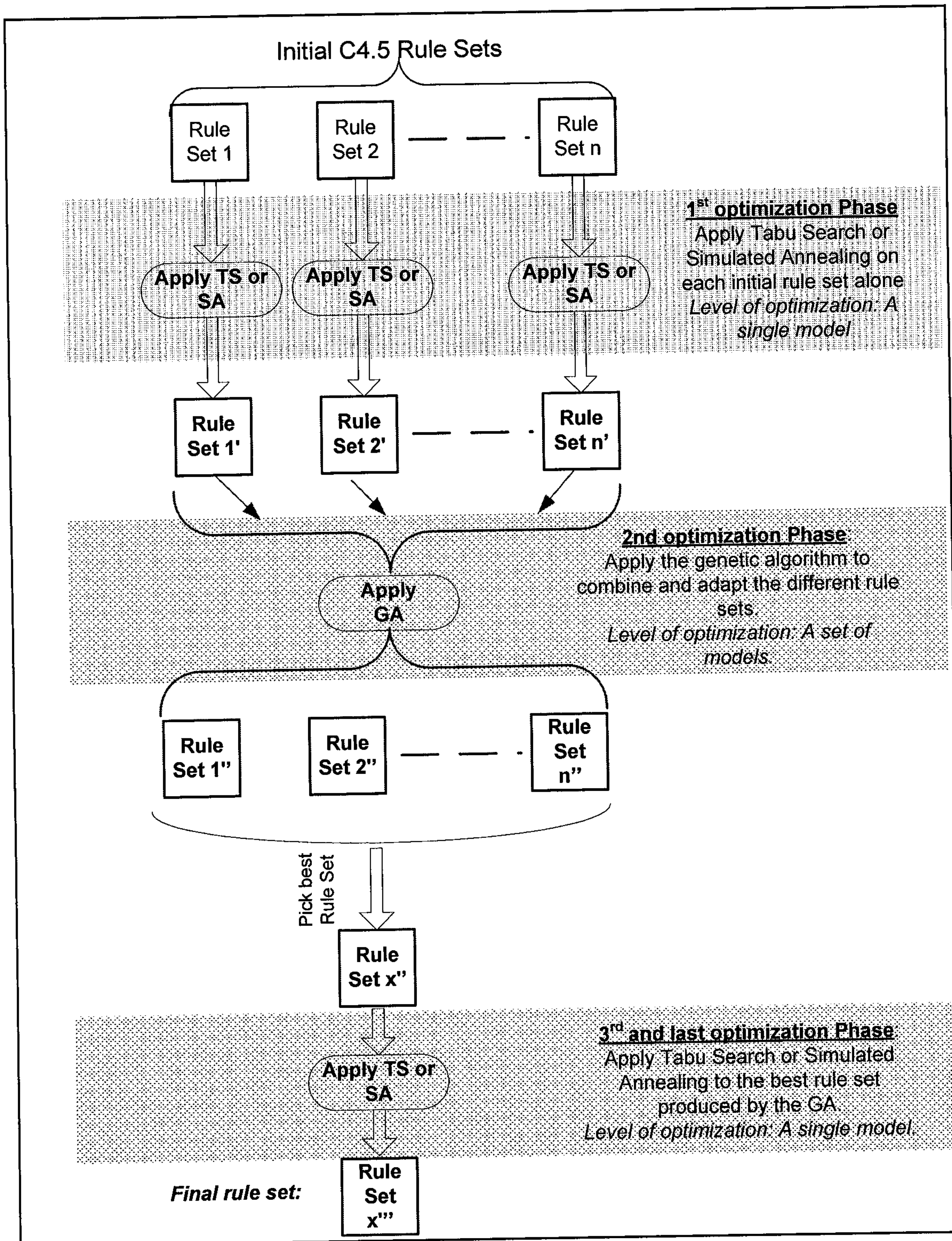


Figure 9: A general functional model describing the hybrid heuristics.

In the next sections, we go through the details of the three main components of the hybrid heuristics: The Genetic Algorithm, the Simulated Annealing and the Tabu Search.

6.1.2 The Genetic Algorithm

The genetic algorithm used in our hybrid heuristics is based on a graphical representation of chromosomes. In our case, each chromosome encodes a rule set. In the next section, we give a thorough overview of the graph representation.

6.1.2.1 The graph representation

The population of chromosomes is represented in a tripartite graph composed of nodes and edges. Nodes of the graph are shown at three levels: The upper level nodes represent the classification labels of the rules. The middle level nodes represent attributes and the bottom level nodes represent values taken by the attributes.

As for edges, we have two kinds of weighted edges: Special edges and normal edges. Normal edges link attribute nodes to value nodes. They represent a condition inside a rule. A weight of 1 represents the operator ">" and the weight of 0 represents the operator "<=". Special edges link attribute nodes to class label nodes. They have a weight of -1 and indicate the classification label of the rule. Note that they link only one attribute node belonging to the rule to the class label node of the rule.

To differentiate each chromosome or rule set, edges belonging to a rule set will have a different color from edges belonging to another. Also, to differentiate each rule inside a rule set from the other we will assign a different line pattern for edges belonging to a particular rule.

The default class label of the rule set is not represented graphically since it does not affect any of the GA operators and does not participate in crossover or mutation.

To illustrate, let us consider the following rule sets:

RULE SET 1

CUB <= 12.0 Stress <= 0.3 class 1

CUB > 10 class 0

NOC > 5 class 1

Stress > 0.6 class 0

RULE SET 2

CUB > 6 class 0

DIT <= 4 NOP >2 class 1

Stress <= 0.4 class 0

Rule set 1 is composed of four rules while rule set 2 is composed of the three rules. Let us assign the color red for rule set 1 and blue for rule set 2. Each rule inside the rule set will have a different line pattern. Figure 10 shows the graph representation obtained.

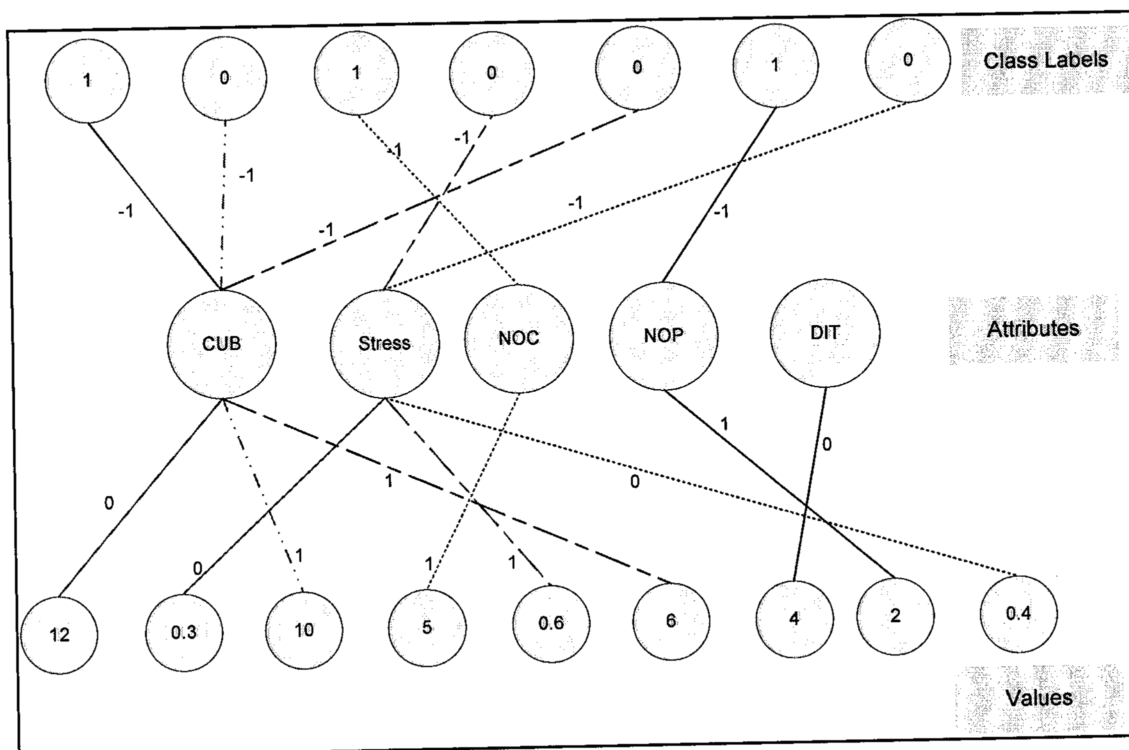


Figure 10: The graph representation.

6.1.2.2 The selection scheme

The selection scheme used in our Genetic Algorithm is rank selection. For more information on this selection scheme, the user can refer to Chapter 3 of this thesis.

6.1.2.3 The fitness function

The fitness function reflects the goodness of the chromosome or the rule set. In our experiments, we used different fitness functions to evaluate the rule sets. Examples of fitness functions used are the accuracy and the J-Index. These evaluation criteria are defined in Chapter 2 of this thesis.

6.1.2.4 The genetic operators

In this section, we show how the main genetic operators, namely crossover and mutation occur in the graphical genetic algorithm.

6.1.2.4.1 Crossover

Crossover consists of selecting two parent chromosomes and exchanging information between them in a way to form two new children chromosomes. Note that crossover between two identical chromosomes is not allowed since no real information exchange would occur in this case. Two types of crossover were implemented: Rule crossover and condition crossover.

6.1.2.4.2 Rule Crossover:

Rule crossover consists of swapping the colors of the edges indicated with a * in Figure 11. Figure 12 shows the graph obtained after crossover. This is equivalent to choosing a random number of rules from the first parent rule set and a random number of rules from the second parent rule set and swapping them. The order of the rules inside the children rule sets is chosen randomly. Figure 11 shows the parent chromosomes and encodes the following rule sets:

PARENT 1

CUB <= 12.0 Stress <= 0.3 class 1

CUB > 10 class 0

NOC > 5 class 1

Stress > 0.6 class 0

PARENT 2

CUB > 6 class 0

DIT <= 4 NOP >2 class 1

Stress <= 0.4 class 0

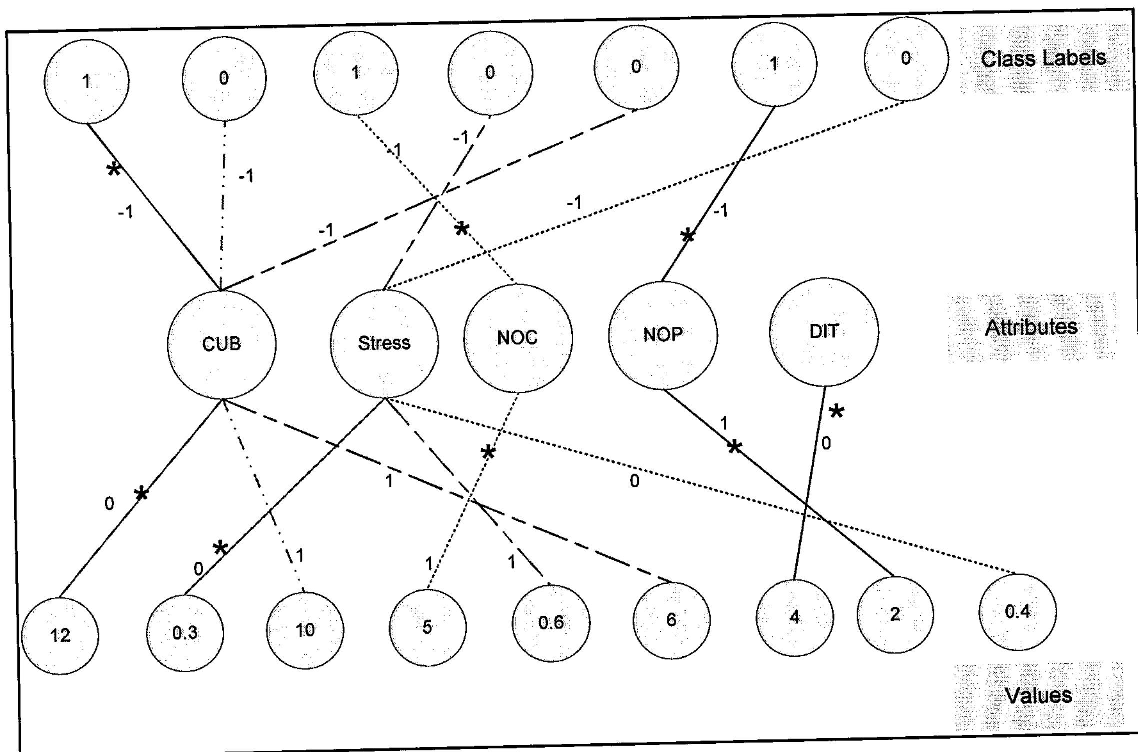


Figure 11: Parents before rule crossover (each chromosome is identified by a color).

The graphical representation of the children chromosomes can be seen in Figure 12. This graph encodes the following rule sets:

CHILD 1

CUB > 10 class 0

DIT <= 4 NOP >2 class 1

Stress > 0.6 class 0

CHILD 2

CUB > 6 class 0

NOC > 5 class 1

CUB <= 12.0 Stress <= 0.3 class 1

Stress <= 0.4 class 0

Hence crossover is equivalent to exchanging the 1st and 3rd rules from parent 1 with the 2nd rule from Parent 2 .

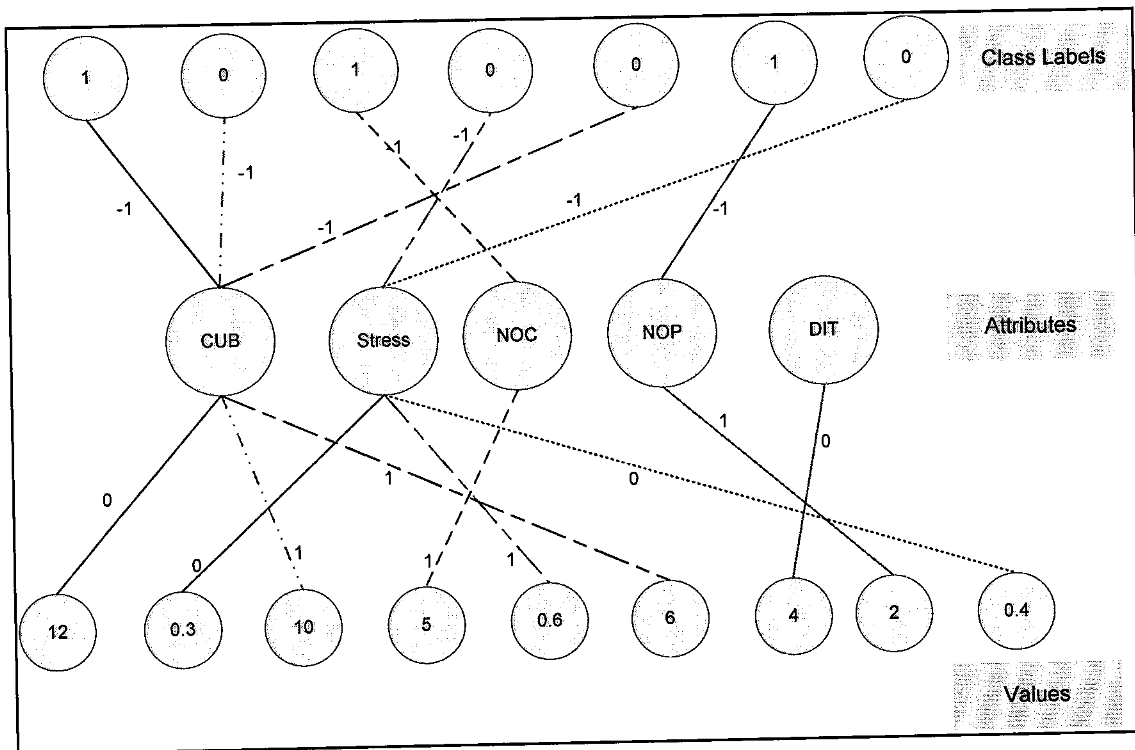


Figure 12: The children after rule crossover.

6.1.2.4.3 Condition Crossover:

Condition crossover consists of swapping the colors between two edges in the same graph. The edges should initially be of different colors. Figures 13 and 14 show the parent chromosomes and the resulting offspring respectively.

This results in the swapping of conditions in two different rule sets. For example, condition crossover on Parent 1 and Parent 2 below results in the two rule sets indicated Child1 and Child2 as shown next.

PARENT 1

CUB <= 12.0 **Stress <= 0.3** class 1

CUB > 10 class 0

NOC > 5 class 1

Stress > 0.6 class 0

PARENT 2

CUB > 6 class 0

DIT <= 4 NOP >2 class 1

Stress <= 0.4 class 0

CHILD 1

CUB <= 12.0 **CUB > 6** class 1

CUB > 10 class 0

NOC > 5 class 1

Stress > 0.6 class 0

CHILD 2

Stress <= 0.3 class 0

DIT <= 4 NOP >2 class 1

Stress <= 0.4 class 0

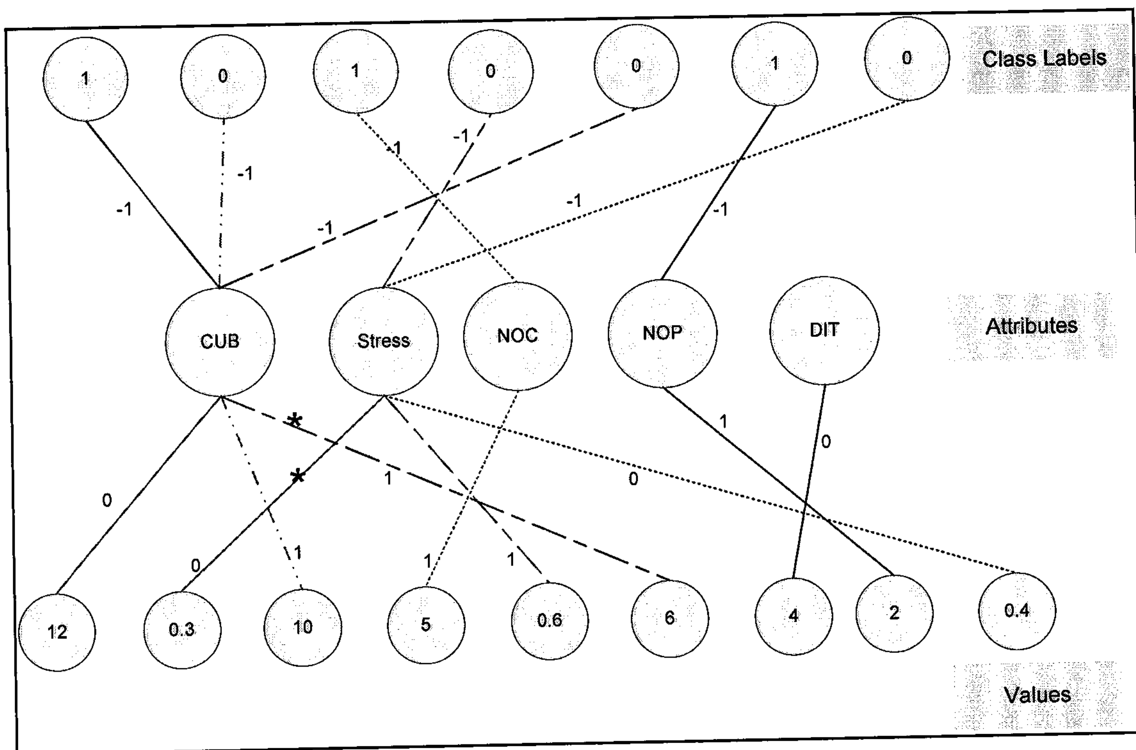


Figure 13: Parents before condition crossover (Edges chosen are labeled with a *).

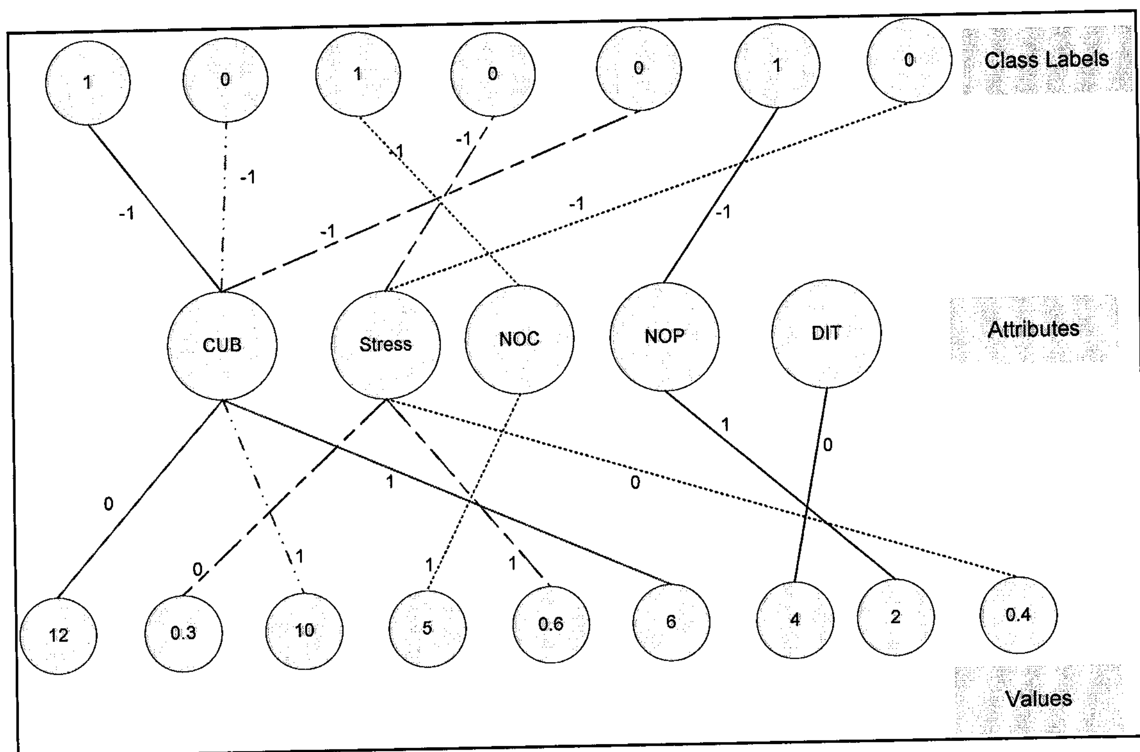


Figure 14: The children after condition crossover.

6.1.2.4.4 Mutation

Before being thrown into the new generation, children chromosomes may go through mutation according to a certain probability. Below we present four mutation operations:

Operator mutation

Operator mutation consists of changing the weight of the normal edge from 1 to 0 or from 0 to 1. This is equivalent to changing the operator of a randomly chosen condition inside the rule set.

To illustrate Figures 15 and 16 show a chromosome before and after operator mutation on edge linking attribute node "Stress" to value node "0.6".

The underlying rule sets are:

Rule Set before operator mutation (Figure 15)

CUB <= 12 CUB > 6 class 1

Stress > 0.6 class 0

NOC <= 5 class 1

Rule Set after operator mutation (Figure 16)

CUB <= 12 CUB > 6 class 0

Stress <= 0.6 class 0

NOC <= 5 class 1

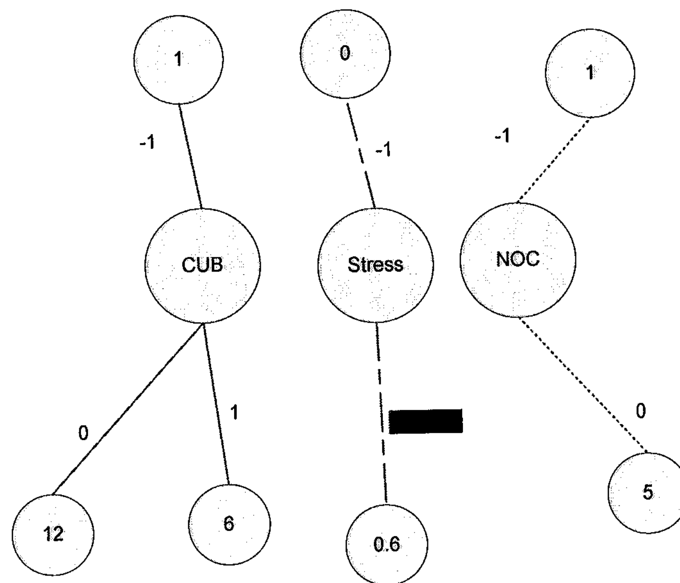


Figure 15: Chromosome before operator mutation on edge linking attribute node "Stress" to value node "0.6".

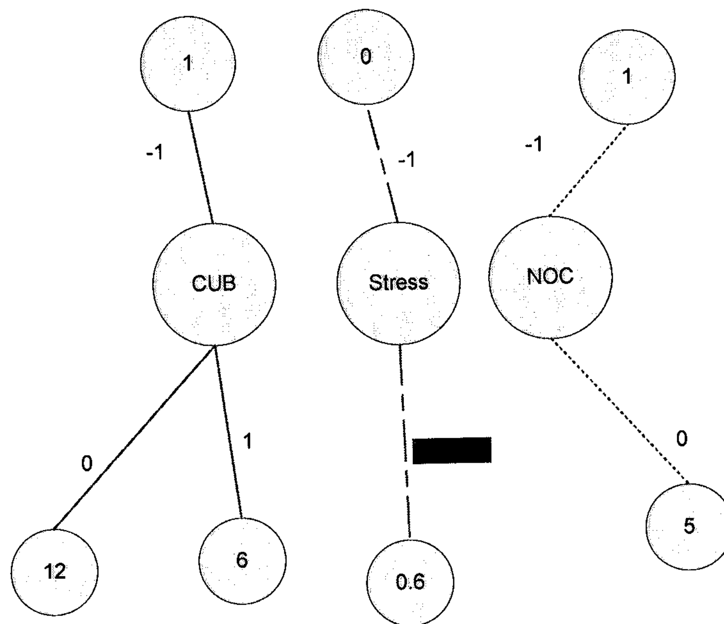


Figure 16: Chromosome after operator mutation on edge linking attribute node "Stress" to value node "0.6".

Class mutation

Class mutation consists of changing the value of a class label node. This is equivalent to changing the class label of a randomly chosen rule inside the rule set.

Figures 17 and 18 show a chromosome before and after class mutation. The underlying rule sets are:

Rule Set before class mutation (Figure 17)

CUB <= 12 CUB > 6 class 1

Stress > 0.6 class 0

NOC <= 5 class 1

Rule Set after class mutation (Figure 18)

CUB <= 12 CUB > 6 class 0

Stress > 0.6 class 0

NOC <= 5 class 1

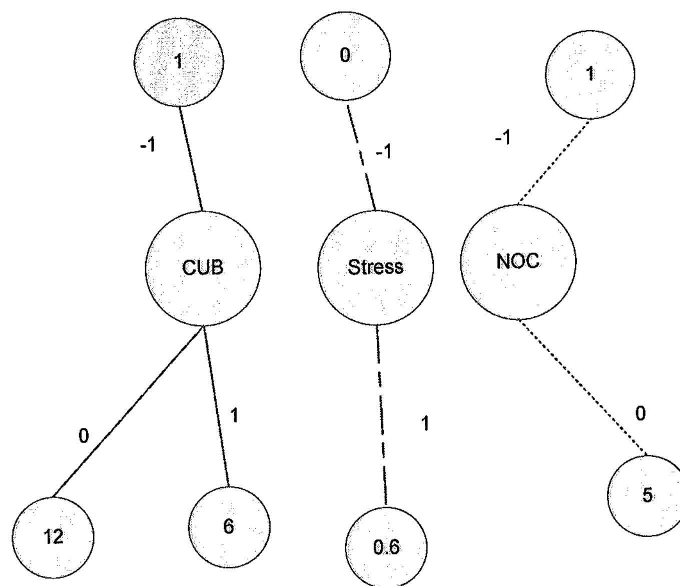


Figure 17: Chromosome before class mutation (the yellow node indicates the mutation point).

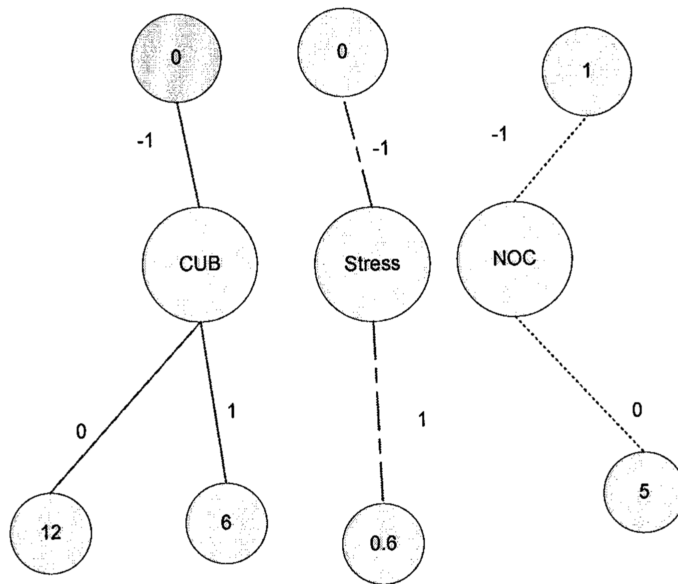


Figure 18: Chromosome after class mutation (The yellow node indicates the vertex that was affected by mutation).

Value mutation

Value mutation consists of randomly changing a value in a value node[1]. This is equivalent to choosing a condition inside a rule set and changing the value of the numerical operand. Figures 19 and 20 show a chromosome before and after a value mutation.

The underlying rule sets are:

Rule Set before value mutation (Figure 19)

CUB <= 12 CUB > 6 class 1

Stress > 0.6 class 0

NOC <= 5 class 1

Rule Set after value mutation (Figure 20)

CUB <= 12 CUB > 6 class 1

Stress > 0.6 class 0

NOC <= 4 class 1

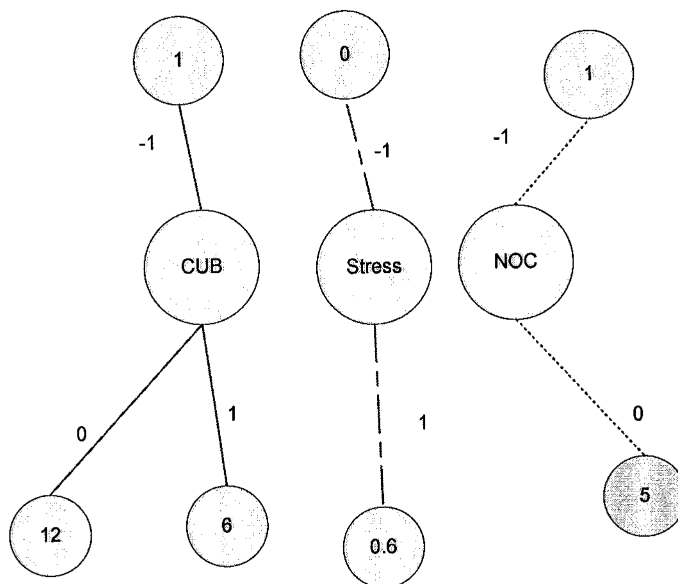


Figure 19: Chromosome before value mutation occurring on the yellow vertex.

[1] The new value is chosen from the set of allowed values for the corresponding attribute. These contain the set of values that the attribute is taking in the dataset.

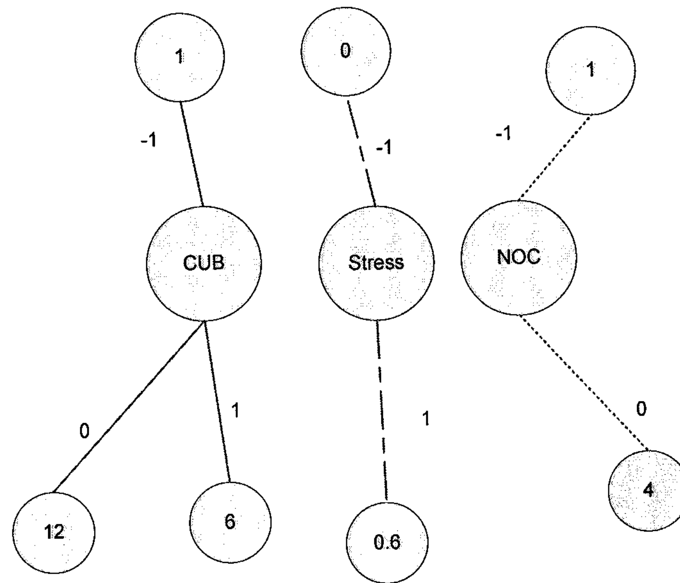


Figure 20: Chromosome after value mutation occurring on yellow vertex (the value was changed from 5 to 4).

Condition addition mutation

Condition addition mutation consists of inserting a new edge in the graph with a line pattern chosen from the set of line patterns already present in the graph. In fact the line pattern of the edge will determine the rule to which the new condition is inserted. The new edge is created by choosing a random weight, a random attribute node and a random value node where the value is picked from the set of allowed values of the attribute.

Figures 21 and 22 show a chromosome before and after condition addition mutation.

The underlying rule sets are:

Rule Set before condition addition mutation (Figure 21)

CUB <= 12 CUB > 6 class 1

Stress > 0.6 class 0

NOC <= 5 class 1

Rule Set after condition addition mutation (Figure 22)

CUB <= 12 CUB > 6 class 1

Stress > 0.6 class 0

NOC <= 4 DIT > 3 class 1

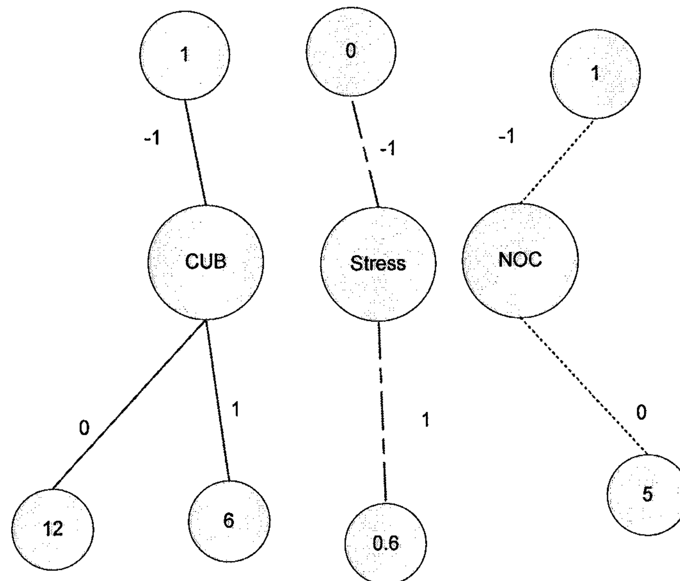


Figure 21: Chromosome before condition addition mutation.

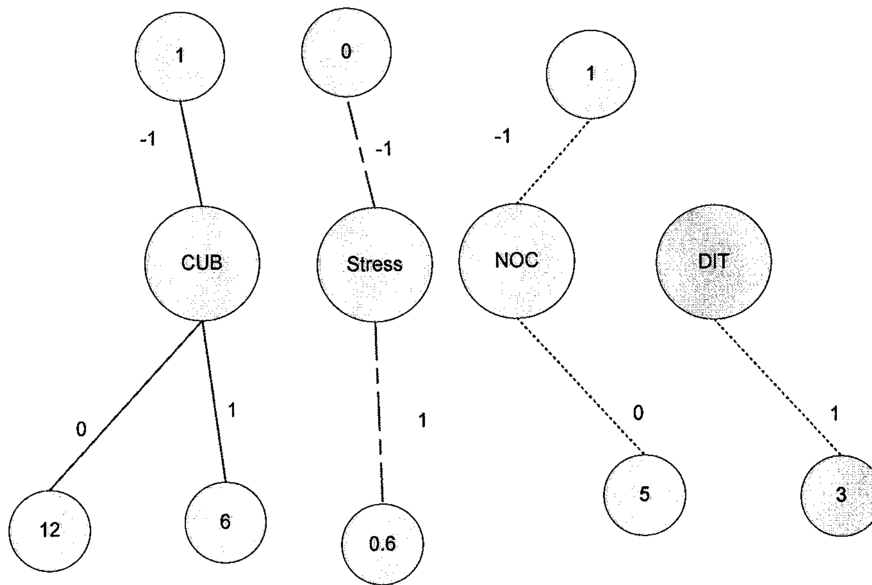


Figure 22: Chromosome after adding the two nodes shown in yellow and the edge that connects them.

6.1.2.4.5 Elitism

Elitism consists of picking the fittest x% chromosomes and copying them directly to the new generation. In our GA, elitism is used with a rate of 10%.

6.1.2.5 Post-Processing and Trimming

Before throwing in the children to the next generation, the GA eliminates redundancy and inconsistency from the rule sets. Redundancy occurs when one condition implies another, when two rules have the same conditions or when two rule sets have the same rules. Inconsistency occurs when two conditions inside a rule are contradictory making the rule unclassifiable (e.g DIT > 3 and DIT <1)

6.1.2.5.1 Removing redundancy:

Removing redundancy is done at three levels, in the following order:

Condition level:

At the condition level, we aim to remove redundant conditions inside every rule. In fact, if a condition implies another we have to remove the latter. We also have to remove duplicate conditions.

For instance, in the following rule *"DIT <= 4.0 DIT <= 2 Stress > 0 class 0"* we see that condition *"DIT <= 2"* implies the condition *"DIT <=4.0"*. So we remove condition *DIT <=4.0* and the rule becomes *"DIT <= 2 Stress > 0 class 0"* . We do not remove the implying condition "DIT <=2" because it affects the semantics of the rule and thus its accuracy and we are only removing redundancy thus generating a rule that is equivalent to the previous one.

Rule level:

At the rule level, we remove identical rules inside a rule set. Two rules are identical if they have the same conditions and the same classification label. Note that the order of the conditions does not matter.

In fact, if rule "*DIT <= 2 Stress > 0.0 class 0*" and rule "*Stress > 0.0 DIT <= 2 class 0*" are inside the same rule set then the redundancy removal method will remove one of them

Rule Set level:

At the rule set level, the redundancy removal method will compare every two rule sets in the final population and check if they are identical. Two rule sets are identical if they have the same rules (with the same order) and the same default classification label.

6.1.2.5.2 Removing inconsistency:

Inconsistency occurs when a rule has two contradictory conditions (conditions that cannot be true at the same time). An inconsistent rule will be eliminated from the rule set.

6.1.2.6 The termination condition

The termination condition in our GA is defined by a certain number of generations. In all the experiments the number of generations will be set to 500 because we observed that no improvement is made at a higher number of generations.

6.1.3 The Simulated Annealing Algorithm

In simulated annealing, the goal is to improve a set of solutions by following a sequence of "perturbations" and comparing the old solution (before perturbation) with the new one (after perturbation) following a

certain evaluation function that determines the quality of the solution. In our case, each solution encodes a rule set.

The simulated annealing algorithm is applied to a single rule set to adapt it to a dataset. More precisely, the algorithm is the same as the one shown in section IV-2 of this thesis.

6.1.3.1 The evaluation function

The evaluation function determines the goodness of a rule set. Evaluation functions used in our experiments are the accuracy and the J-Index, or a linear combination of both.

6.1.3.2 The perturbation function

We implemented six types of perturbation functions with equal probabilities for a rule set.

Rule Deletion:

This consists of choosing randomly a rule inside the rule set and removing it from the rule set.

Condition Deletion:

This is consists of choosing randomly a rule inside the rule set and deleting one of its conditions in a random manner.

Rule Addition:

This consists of creating a new random rule and inserting it into the rule set at a random position.

Condition Addition:

This consists of creating a random condition and inserting it to one of the rules of the rule set.

Rule Change:

This consists of choosing a rule from the rule set and replacing it with a new randomly created one.

Condition Change:

This consists of choosing randomly a condition inside a rule from the rule set and replacing it with a new randomly created condition.

6.1.4 The Tabu Search Algorithm

In the two heuristics TS-GA and SA-TS-GA, Tabu search is used to individually optimize the initial c4.5 rule sets and the final rule set generated by the GA. Similarly to Simulated Annealing, in the tabu search algorithm, each solution encodes a rule set. For further details on the algorithm, the user can refer to section V.2.2 of this thesis.

6.1.4.1 The evaluation function:

The evaluation function reflects the quality of the rule set. In our experiments we use three different evaluation functions. They are the accuracy, the J-Index or a linear combination of both. The choice of the evaluation function is different for each experiment as it depends on the nature of the dataset. For example, in an imbalanced dataset the J-Index is used because it is a better indicator of the accuracy of the heuristic on minority classification labels.

6.1.4.2 The Neighborhood function:

A neighbor of a solution S is defined as a rule set that is **not on the tabu list** and that differs from S either by a rule or by a condition. We have designed six different transformations that create neighbor solutions. They all occur with equal probability. They are:

Rule Deletion:

This consist of removing a random rule from the rule set

Condition Deletion:

This consists of to choosing randomly a rule inside the rule set and deleting one of its conditions chosen randomly.

Rule Addition:

This consists of creating a new random rule and inserting it into the rule set at a random position.

Condition Addition:

This is consists of creating a random condition and inserting it into one of the rules of the rule set (the rule is also chosen randomly).

Rule Change:

This consists of choosing a random rule from the rule set and replacing it with a new randomly created one.

Condition Change:

This consists of choosing randomly a condition inside a rule from the rule set and replacing it with a new randomly created condition.

6.1.4.3 The Tabu List Replacement Policy

Since the number of iterations used for each rule set is relatively small, a solution remains in the tabu list for 10 iterations. Also, we observed that after ten iterations, the search does not cycle back to a previously visited solution because of the diverse alterations and modifications triggered by the neighborhood function.

6.2 The experiments

In this section, we describe the experiments conducted in order to evaluate the performance of our three hybrid heuristics.

6.2.1 The Datasets

In order to assess the performance of the hybrid heuristics, we tested them on two datasets taken from the stability of object-oriented software components. These datasets, namely STAB1 and STAB2 were also used in (Azar, 2004).

STAB1

STAB1 was formed by the extraction of nineteen structural metrics (Table 2) from the eleven software systems shown in Table 3 using the ACCESS tool and the Discover environment © . Fifteen subsets were created by combining these metrics and were used with the 11 software systems shown in Table 3 to create 165 data sets. C4.5 was then used to create 165 decision tree classifiers from these data sets. Classifiers having a single classification label and classifiers with a training error more than 10% were eliminated and the remaining 40 were kept. Rule sets were then extracted from the decision trees.

The three heuristic techniques we developed use these rule sets and adapt them to the four software systems shown in Table 4. These form the STAB1 data set which consists of 2920 instances.

Name	Description
Cohesion metrics	
LCOM	lack of cohesion on methods
COH	cohesion
COM	cohesion metric
COMI	cohesion metric inverse
Coupling metrics	
OCMAIC	other class method attribute import coupling
OCMAEC	other class method attribute export coupling
CUB	number of classes used by a class
Inheritance metrics	
NOC	number of children
NOP	number of parents
DIT	depth of inheritance
MDS	message domain size
CHM	class hierarchy metric
Size complexity metrics	
NOM	number of methods
WMC	weighted methods per class
WMC_LOC	LOC weighted methods per class
MCC	McCabe's complexity weighted methods per class
NPPM	number of public and protected methods in a class
NPA	number of public attributes
The Stress metric	
Stress	stress applied to a class

Table 2: STAB1 software quality metrics used as attributes in the classifiers (Azar, 2004).

Software System	Number of Versions (major)	Number of classes
Bean browser	6(4)	388-392
Ejbvoyager	8(3)	71-78
Free	9(6)	46-93
Javamapper	2(2)	18-19
Jchempaint	2(2)	84
Jedit	2(2)	464-468
Jetty	6(3)	229-285
Jigsaw	4(3)	846-958
Jlex	4(2)	20-23
Lmjs	2(2)	106
Voji	4 (4)	16-39

Table 3: STAB1-Software systems used to build classifiers with C4.5 (Azar, 2004).

JDK version	Number of classes
jdk1.0.2	187
jdk1.1.6	583
jdk1.2.004	2337
jdk1.3.0	2737

Table 4: STAB1- Software systems used to train and test the heuristics (Azar, 2004).

STAB2 data collection

STAB1 is a very imbalanced dataset. In fact, 2481 cases are stable and 439 unstable. In order to test our hybrid heuristics on a balanced data set, we used STAB2 which is also taken from the stability of object oriented software components. The 22 metrics used are shown in Table 5. Fifteen subsets of metrics were created by combining 2, 3, or 4 groups in all possible ways. These subsets were used with the 9 software systems shown in Table 6 to create 135 data sets. C4.5 was used to construct a decision tree from each data set. 23 classifiers were retained after the elimination of constant classifiers (classifiers having a single classification label) and classifiers with an error rate higher than 10%. C4.5 was then used to covert the decision trees to rule sets. The systems shown in Table 7 are used to train and test our three heuristic techniques. These form the STAB2 data set.

Name	Description
Cohesion metrics	
LCOM	lack of cohesion on methods
COH	cohesion
COM	cohesion metric
COMI	cohesion metric inverse
Coupling metrics	
OCMAIC	other class method attribute import coupling
OCMAEC	other class method attribute export coupling
CUB	number of classes used by a class
CUBF	number of classes used by a member function
Inheritance metrics	
NOC	number of children
NOP	number of parents
NON	number of nested classes
NOCONT	number of containing classes
DIT	depth of inheritance
MDS	message domain size
CHM	class hierarchy metric
Size complexity metrics	
NOM	number of methods
WMC	weighted methods per class
WMC_LOC	LOC weighted methods per class
MCC	McCabe's complexity weighted methods per class
DEPCC	operation access metric
NPPM	number of public and protected methods in a class
NPA	number of public attributes.

Table 5: STAB2-Software quality metrics used as attributes in the classifiers (Azar, 2004).

Software System	Number of Versions (major)	Number of classes
Bean browser	6(4)	388-392
Ejbvoyager	8(3)	71-78
Free	9(6)	46-93
Javamapper	2(2)	18-19
Jchempaint	2(2)	84
Jigsaw	4(3)	846-958
Jlex	4(2)	20-23
Lmjs	2(2)	106
Voji 4	4(4)	16-39

Table 6: STAB2-Software systems used to build classifiers with C4.5 (Azar, 2004).

Software System	Number of Versions (major)	Number of classes
Jedit	2(2)	464-468
Jetty	6(3)	229-285

Table 7: STAB2-Software systems used to train and test the heuristics (Azar, 2004).

6.2.2 Experimental Settings

A total of three experiments were done on each of the three hybrid heuristics. To accurately assess the performance of each hybrid heuristic compared to *C4.5*, we performed 10-cross-validation on each of STAB1 and STAB2 datasets. In this technique, the dataset is divided into ten roughly equal blocks: One of the ten blocks is chosen to be used as the testing test and the remaining nine are combined to form the training set. This process is repeated 10 times, each time using a different block for the testing set. At the end we obtain ten different "folds" of training/testing sets.

Moreover, to account for the randomness factor in the heuristics, each experiment is repeated for a certain number of runs. We used 10 runs for experiments on STAB2 and 5 runs for the experiment on STAB1. At the end of each run, the best rule set obtained by the heuristic is retrieved and its accuracy and J-Index on training and testing sets is reported. The final results are obtained by calculating the average and

standard deviation of the accuracy and the J-Index over the 10 folds. The heuristic's results are then compared with the average and standard deviation over the 10 folds of the accuracy and J-Index of the best C4.5 rule set. The "best" rule set is the one that has the highest evaluation function. We will use different evaluation functions for each experiment.

More precisely, Table 8 shows the settings and parameters used in the experiments with the SA-GA heuristic; Table 9 shows the parameters used for the TS-GA heuristic and Table 10 shows those used for the SA-TS-GA heuristic.

We chose to do 10 runs for experiments on STAB2 in order to account for the randomness factor in the heuristics. Due to limited technical resources and STAB1 being a dataset three times larger than STAB2, we could only do 5 runs for experiment 3. However this number of runs is enough since the standard deviation of the results is small.

The parameters of the Simulated Annealing, Tabu Search and Genertic Algorithm were chosen as optimal after testing several different parameters. In the GA, we chose to give a probability for rule crossover equal to 80%, a probability of for condition crossover equal to 10% and a mutation rate of 10%. The rule crossover is given a higher probability than condition crossover because since it involves the biggest information exchange between the chromosomes.

The main differences between the three experiments are the following:

- The choice of the dataset: Experiments 1 and 2 are done on STAB2 while experiment 3 is done on STAB1.
- The choice of the evaluation/fitness function: Since STAB2 is a balanced dataset, we focused on improving the accuracy thus giving it more weight in the fitness/evaluation functions: More precisely, in a first experiment we use *f=accuracy* as fitness/evaluation function and in a second experiment we use *f* =

*accuracy +0.5*J-Index* as accuracy/evaluation function. STAB1 being an imbalanced dataset, we aimed to improve the J-Index by setting it as fitness/evaluation function.

General Parameters	
Number of runs for experiments 1 and 2 done on STAB 2	10
Number of runs for experiment 3 on STAB 1	5
Simulated Annealing Parameters	
Number of iterations for each initial rule set (1st optimization phase)	10
Number of iterations per temperature for each initial rule set (1st optimization phase)	10
Number of iterations for final rule set (3rd optimization phase)	100
Number of iterations per temperature for final rule set (3rd optimization phase)	100
Initial temperature T0	0.7
Boltzmann constant K	0.5
Evaluation function for experiment 1	f =accuracy
Evaluation function for experiment 2	f =accuracy + 0.5*J-Index
Evaluation function for experiment 3	f = J-Index
Genetic Algorithm Parameters	
Rule Crossover Probability	80%
Condition Crossover Probability	10%
Operator Mutation Probability	2.5%
Value Mutation Probability	2.5%
Class Mutation Probability	2.5%
Condition Addition Mutation Probability	2.5%
Number of generations	500
Fitness function for experiment 1	f =accuracy
Fitness function for experiment 2	f =accuracy + 0.5*J-Index
Fitness function for experiment 3	f = J-Index

Table 8: Settings and parameters for the SA-GA hybrid heuristic.

General Parameters	
Number of runs for experiments 1 and 2 done on STAB 2	10
Number of runs for experiment 3 done on STAB 1	5
Tabu Search Parameters	
Number of iterations for each initial rule set	100
Number of iterations for final rule set (3rd optimization phase)	100
Tabu List size	10
Evaluation function for experiment 1	f =accuracy
Evaluation function for experiment 2	f =accuracy + 0.5*J-Index
Evaluation function for experiment 3	f = J-Index
Genetic Algorithm Parameters	
Rule Crossover Probability	80%
Condition Crossover Probability	10%
Operator Mutation Probability	2.5%
Value Mutation Probability	2.5%
Class Mutation Probability	2.5%
Condition Addition Mutation Probability	2.5%
Number of generations	500
Fitness function for experiment 1	f =accuracy
Fitness function for experiment 2	f =accuracy + 0.5*J-Index
Fitness function for experiment 3	f = J-Index

Table 9: Settings and parameters for the TS-GA hybrid heuristic.

General Parameters	
Number of runs for experiments 1 and 2 done on STAB 2	10
Number of runs for experiment 3 done on STAB 1	5
Tabu Search Parameters	
Number of iterations for each initial rule set	100
Number of iterations for final rule set (3rd optimization phase)	100
Tabu List size	10
Evaluation function for experiment 1	f =accuracy
Evaluation function for experiment 2	f =accuracy + 0.5*J-Index
Evaluation function for experiment 3	f = J-Index
Simulated Annealing Parameters	
Number of iterations for each initial rule set	10
Number of iterations per temperature for each initial rule set	10
Number of iterations for final rule set (3rd optimization phase)	100
Number of iterations per temperature for final rule set (3rd optimization phase)	100
Initial temperature T0	0.7
Boltzmann constant K	0.5
Evaluation function for experiment 1	f =accuracy
Evaluation function for experiment 2	f =accuracy + 0.5*J-Index
Evaluation function for experiment 3	f = J-Index
Genetic Algorithm Parameters	
Rule Crossover Probability	80%
Condition Crossover Probability	10%
Operator Mutation Probability	2.5%
Value Mutation Probability	2.5%
Class Mutation Probability	2.5%
Condition Addition Mutation Probability	2.5%
Number of generations	500
Fitness function for experiment 1	f =accuracy
Fitness function for experiment 2	f =accuracy + 0.5*J-Index
Fitness function for experiment 3	f = J-Index

Table 10: Settings and parameters for SA-TS-GA hybrid heuristic.

6.2.3 Experimental Results and Analysis

This section presents the comparative results of the three heuristics in the three experiments.

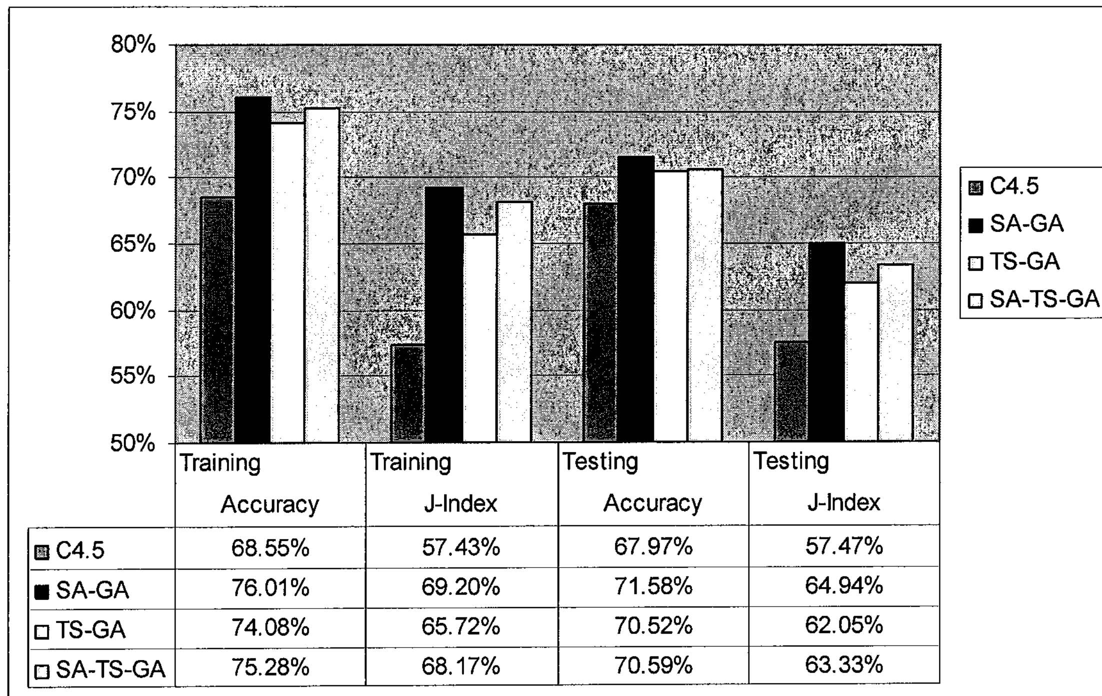
6.2.3.1 Experiment 1: Done on STAB2 using accuracy as the evaluation function.

The first experiment was done on the balanced STAB2 dataset using the accuracy as evaluation/fitness function.

Table 11 shows the results of the three hybrid heuristics compared to C4.5 with standard deviation shown in parentheses. Figure 23 shows the corresponding bar chart.

	Accuracy Training	J-Index Training	Accuracy Testing	J-Index Testing
C4.5	68.55% (0.7%)	57.43% (0.5%)	67.97% (5.57%)	57.47% (3.50%)
SA-GA	76.01% (0.54%)	69.20% (1.08%)	71.58% (5.04%)	64.94% (4.67%)
TS-GA	74.08% (0.56%)	65.72% (0.95%)	70.52% (5.53%)	62.05% (3.93%)
SA-TS-GA	75.28% (0.41%)	68.17% (1.13%)	70.59% (5.38%)	63.33% (4.50%)

Table 11: Results of the three hybrid heuristics compared to C4.5 in Experiment 1 done on STAB2. Standard deviation is shown in parenthesis.



	Training Accuracy	Training J-Index	Testing Accuracy	Testing J-Index
C4.5	68.55%	57.43%	67.97%	57.47%
SA-GA	76.01%	69.20%	71.58%	64.94%
TS-GA	74.08%	65.72%	70.52%	62.05%
SA-TS-GA	75.28%	68.17%	70.59%	63.33%

Figure 23: Bar Chart showing the results of the three hybrid heuristics compared to C4.5 in Experiment 1 done on STAB2.

Concerning the accuracy, all three hybrid heuristics were able to outperform C4.5 on both training and testing sets. However, the SA-GA heuristic made the highest improvement with an increase in accuracy of 7.5% on training sets and 3.6% on testing sets. The SA-TS-GA comes second with an average improvement of 6.7% on training sets and 2.6% on testing sets, and the TS-GA achieved an improvement of 5.5% on training sets and 2.6% on testing sets.

Concerning the J-Index, all three heuristics were able to improve it although it was not set as evaluation/fitness function in this experiment. The increase in J-Index means that our heuristics have a better accuracy than *C4.5* on cases with a minority classification. The SA-GA is also the best performer with an average J-Index improvement of 11.8% on training sets and 7.5% on testing sets. The SA-TS-GA comes next with an

average improvement of 10.7% on training sets and 5.9% on testing sets. The TS-GA achieves an improvement with a J-Index increase of 8.2% on training sets and 4.6% on testing sets.

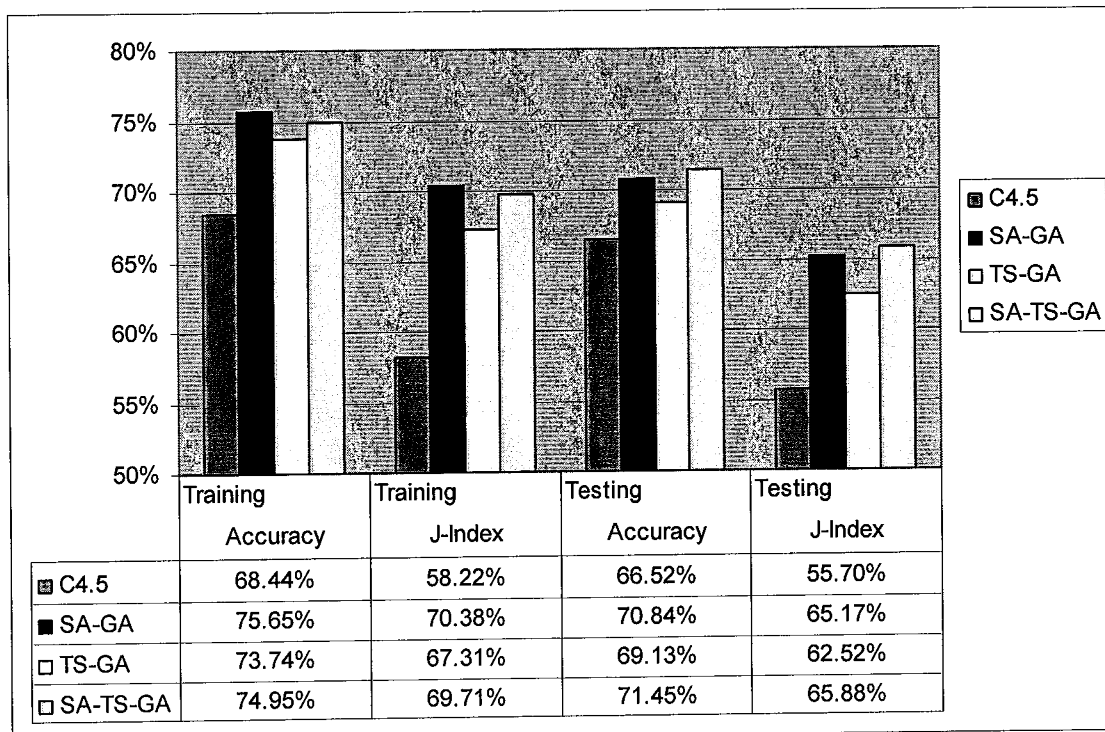
*6.2.3.2 Experiment 2: Done on STAB2 using accuracy + 0.5*J-Index as the evaluation function.*

The second experiment was done on the balanced dataset STAB2. However, this time we chose to give some weight to the J-Index by including it in the evaluation/fitness function. Moreover, since STAB2 is a balanced dataset, the accuracy is the main criteria we wish to improve. Hence, the evaluation/fitness function was set to "accuracy + 0.5*J-Index" as it gives more weight to the accuracy.

Table 12 shows the results of the three hybrid heuristics compared to C4.5 with standard deviation shown in parenthesis. Figure 24 shows the corresponding bar chart.

	Accuracy Training	J-Index Training	Accuracy Testing	J-Index Testing
C4.5	68.44% (0.7%)	58.22% (1.12%)	66.52% (7.56%)	55.70% (3.34%)
SA-GA	75.65% (0.53%)	70.38% (0.97%)	70.84% (5.3%)	65.17% (5.27%)
TS-GA	73.74% (0.66%)	67.31% (1.24%)	69.13% (4.92%)	62.52% (3.82%)
SA-TS-GA	74.95% (0.55%)	69.71% (0.74%)	71.45% (4.47%)	65.88% (4.78%)

Table 12: Results of the three hybrid heuristics compared to C4.5 in Experiment 2 on STAB2. Standard deviation is shown in parenthesis.



	Training Accuracy	Training J-Index	Testing Accuracy	Testing J-Index
C4.5	68.44%	58.22%	66.52%	55.70%
SA-GA	75.65%	70.38%	70.84%	65.17%
TS-GA	73.74%	67.31%	69.13%	62.52%
SA-TS-GA	74.95%	69.71%	71.45%	65.88%

Figure 24: Bar Chart showing the results of the three hybrid heuristics compared to C4.5 in Experiment 2 on STAB2.

Regarding the accuracy, all three hybrid heuristics were able to out beat C4.5 on both the training and testing sets. Nonetheless, the SA-GA heuristic made the best improvement on both testing and training sets with an average increase in accuracy of 7.2% on training sets and 4.3% on testing sets. The SA-TS-GA comes second with an average increase in accuracy of 6.5% on training sets and 4.9% on testing sets. Finally, the TS-GA resulted in the least improvement with an average improvement of 5.3% on training sets and 2.6% on testing sets.

As for the J-Index, all three heuristics were able to improve it. The SA-GA achieves an average J-Index improvement of 12.2% on training sets and 9.5% on testing sets. The SA-TS-GA comes next with an average improvement of 11.5% on training sets and 10.2% on testing sets. The

TS-GA achieves an improvement with a J-Index increase of 9.1% on training sets and 6.8% on testing sets.

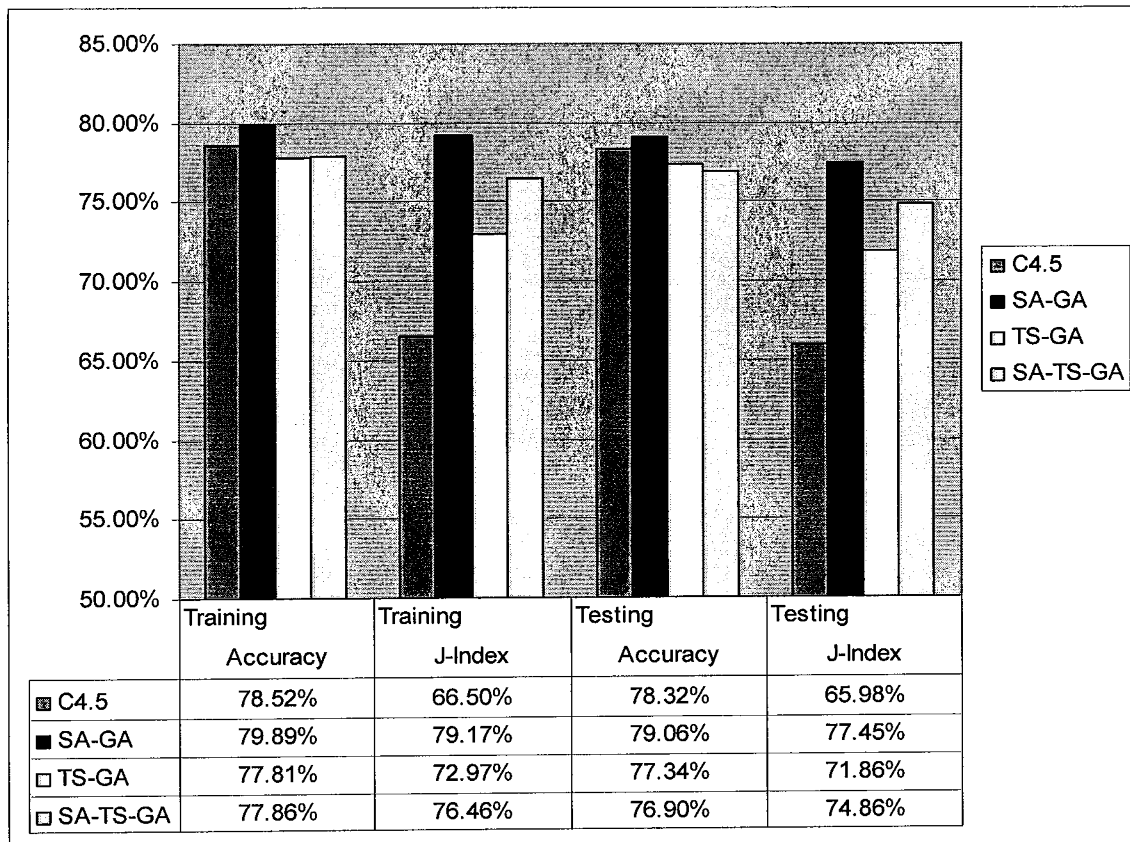
6.2.3.3 Experiment 3: Done on STAB1 using J-Index as the evaluation function

The third experiment was done on STAB1. In an unbalanced dataset, the J-Index is a better performance indicator than the accuracy since it gives more weight to cases with the minority classifier. Since STAB1 is a very imbalanced dataset (84.5% of the instances are stable), we chose to use the J-Index as evaluation/fitness function.

Table 13 shows the results of the three hybrid heuristics compared to C4.5 with standard deviation shown in parentheses. Figure 25 shows the resultant bar chart.

	Accuracy Training	J-Index Training	Accuracy Testing	J-Index Testing
C4.5	78.52% (0.63%)	66.50% (0.5%)	78.32% (1.98%)	65.98% (4.26%)
SA-GA	79.89% (1.84%)	79.17% (0.89%)	79.06% (2.16%)	77.45% (2.17%)
TS-GA	77.81% (1.91%)	72.97% (1.72%)	77.34% (1.79%)	71.86% (5.03%)
SA-TS-GA	77.86% (2.71%)	76.46% (1.82%)	76.90% (2.38%)	74.86% (3.23%)

Table 13: Results of the three hybrid heuristics compared to C4.5 in Experiment 3 on STAB1. Standard deviation is shown in parenthesis.



	Training Accuracy	Training J-Index	Testing Accuracy	Testing J-Index
C4.5	78.52%	66.50%	78.32%	65.98%
SA-GA	79.89%	79.17%	79.06%	77.45%
TS-GA	77.81%	72.97%	77.34%	71.86%
SA-TS-GA	77.86%	76.46%	76.90%	74.86%

Figure 25: Bar Chart showing the results of the three hybrid heuristics compared to C4.5 in Experiment 3 on STAB1.

As we can see, concerning the J-Index, all three heuristics outperformed C4.5. The SA-GA achieves a significant J-Index improvement of 12.7% on training sets and 11.5% on testing sets. The SA-TS-GA achieves an average J-Index improvement of 10% on training sets and 8.9% on testing sets. The TS-GA achieves the smallest improvement with a J-Index increase of 6.5% on training sets and 5.9% on testing sets.

Regarding the accuracy, only the SA-GA was able to slightly improve the initial c4.5 accuracy results with an average improvement of 1.4% on training sets and 0.8% on testing sets. The fact that no real

improvement on the accuracy was made is normal since in this experiment the heuristics are trained to improve the J-Index not the accuracy.

6.2.3.4 Comparison with results obtained in previous work

In this section, we compare the best results achieved by our most performing hybrid heuristic, SA-GA, with previous results obtained by other techniques on the same datasets (STAB1 and STAB2).

Regarding STAB2, the SA-GA was able to outperform the genetic algorithm proposed in (Azar, 2004). In fact, concerning the accuracy, the SA-GA in Experiment 1 was able to achieve an average accuracy of 76% on training sets and 71.6% on testing sets while the genetic algorithm (Azar, 2004) achieved an average accuracy of 74.5% on training sets and 70% on testing sets. Regarding the J-Index, the improvement is more remarkable: SA-GA was able to achieve an average J-Index of 70.4% on training sets and 65.2% on testing sets (Experiment 2) while the genetic algorithm proposed in (Azar, 2004) achieved an average J-Index of 65% on training sets and 60.5% on testing sets. These results are summarized in Table 14. The table also shows that our technique achieved results with smaller standard deviation than (Azar, 2004) results which means that our hybrid heuristic was performing results close to each other in a steady manner. We think that SA-GA was able to outperform previous techniques because of its hybrid nature (previous techniques were based on one heuristic only). In fact, it stands out as being the only technique that:

- works on two optimization levels (the set of rule sets level and the individual rule set level) .
- merges the skills of two powerful heuristic techniques: Simulated Annealing and Genetic Algorithms.

	Accuracy Training	J-Index Training	Accuracy Testing	J-Index Testing
C4.5	68.55% (0.7%)	57.43% (0.5%)	67.97% (5.57%)	57.47% (3.50%)
(Azar, 2004)	74.5% (1%)	65% (3%)	70% (6%)	60.5% (5%)
SA-GA	76.01% (0.54%)	70.38% (0.97%)	71.58% (5.04%)	65.17% (5.27%)
Improvement	**1.5%**	**5.4%**	**1.6%**	**4.7%**

Table 14: Table showing a comparison of SA-GA results with previous results on STAB2. Standard deviation is shown in parenthesis.

Regarding the unbalanced STAB1 dataset, the SA-GA heuristic was able to achieve improvement over previous results regarding the J-Index. In fact, the most performing genetic algorithm of the two genetic algorithms proposed in (Bouktif, Azar, Sahraoui, K´egl & Precup, 2004) achieved an average J-Index of 78.24% on training sets and 76.86% on testing sets while our SA-GA hybrid heuristic achieved an average J-Index of 79.17% on training sets and 77.45% on testing sets (Experiment 3). These results along with standard deviation values are shown in Table 15.

Table 15 also shows that although the overall improvement in J-Index is slight, the SA-GA hybrid heuristic achieved results with smaller standard deviation than the previous techniques results. The small improvement over previous results made by SA-GA might be due to the unbalanced nature of the STAB1 dataset.

	J-Index Training	J-Index Testing
C4.5	66.50% (0.5%)	65.98% (4.26%)
(Bouktif, Azar, Sahraoui, K´egl & Precup, 2004)	78.24 (2.43%)	76.86% (5.06%)
SA-GA	79.17% (0.89%)	77.45% (2.17%)
Improvement	**0.93%**	**0.6%**

Table 15: Table showing a comparison of SA-GA results with previous results on STAB1. Standard deviation is shown in parenthesis.

6.2.3.5 Limitations of the technique

Table 16 and Table 17 show the average number of rules per rule set and the average number of conditions per rule of the rule sets produced by the hybrid heuristics on datasets STAB1 (Table 16) and STAB2 (Table 17). The limitations of the hybrid heuristics reside in the high complexity of the rule sets they produced. In fact, compared to C4.5 and the genetic algorithm of (Azar,2004), the hybrid heuristics' rule sets have a significantly higher number of rules making them more complex. We think that the large number of rules per rule set is due to the *rule crossover* genetic operator of our GA as it can cause the addition of a large number of rules to a rule set in a single operation. The number of conditions per rule is slightly higher than C4.5's with an approximate number of conditions of 3. A rule with three conditions remains simple and can be easily interpreted by human experts.

	Average number of rules per rule set	Average number of conditions per rule
C4.5	3.71	2.1
SA-GA	17.3	2.9
TS-GA	14.2	2.8
SA-TS-GA	16	2.9
(Azar, 2004)	4.9	1.6

Table 16: Size of rule sets on STAB1.

	Average number of rules per rule set	Average number of conditions per rule
C4.5	3.9	2
SA-GA	14.9	2.7
TS-GA	14.3	2.6
SA-TS-GA	13.7	2.7
(Azar, 2004)	3.9	2

Table 17: Size of rule sets on STAB2.

Another limitation is the order of complexity which is $O(n^2)$ and the high runtime of the three hybrid heuristics.

6.2.3.6 Insight on the tri-phased optimization process of the hybrid heuristics

In this section, we examine closely the tri-phased optimization process of the hybrid heuristics. As previously depicted in Figure 9, each hybrid heuristics works in a three-phase optimization scheme. We aim to shed light on the degree of optimization taking place at each phase. For this, we record the average increase in accuracy or J-Index of a rule set at the end of each phase to better understand the inner mechanism of our hybrid heuristics.

To illustrate, Figure 26 shows the increase in accuracy at the end of each optimization phase for each hybrid heuristic during experiment 1 (on STAB2). Figure 27 shows the increase in J-Index at the end of each optimization phase for each hybrid heuristic during experiment 3 (on STAB1). To build the plots, ten different sample runs are chosen randomly for each of the hybrid heuristics and the average over these runs is calculated: The values are shown in the plot. We can see that for all the heuristics, it is during the second optimization phase (the genetic algorithm phase) that the most remarkable improvement is attained. This can be explained by the strong combinational power of the GA as it is the only heuristic that optimizes a set of rule sets by combining them and adapting them to the dataset while the other two heuristics (SA and TS) only adapt a single rule set to the dataset.

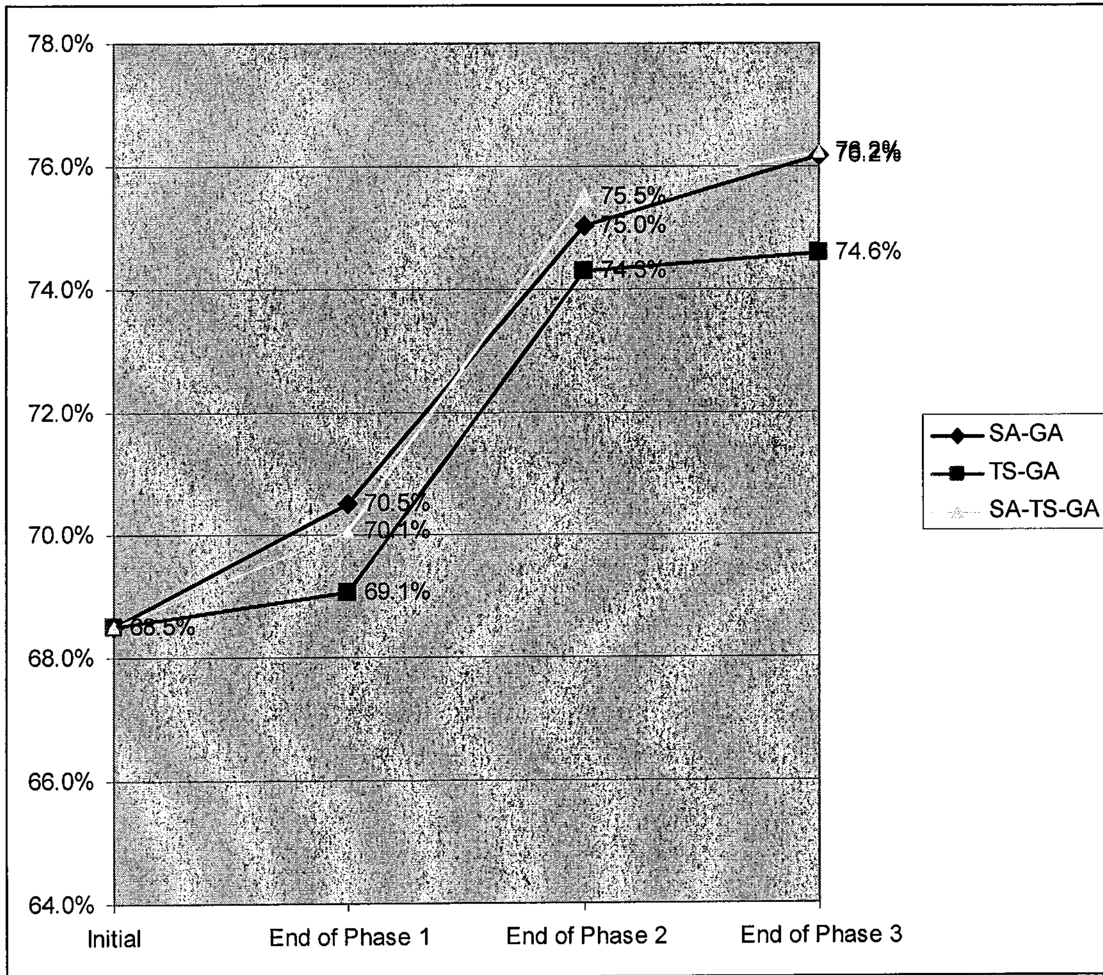


Figure 26: Plot showing the improvement in accuracy in Experiment 1 on STAB2 at the three phases of optimization for each hybrid heuristic.

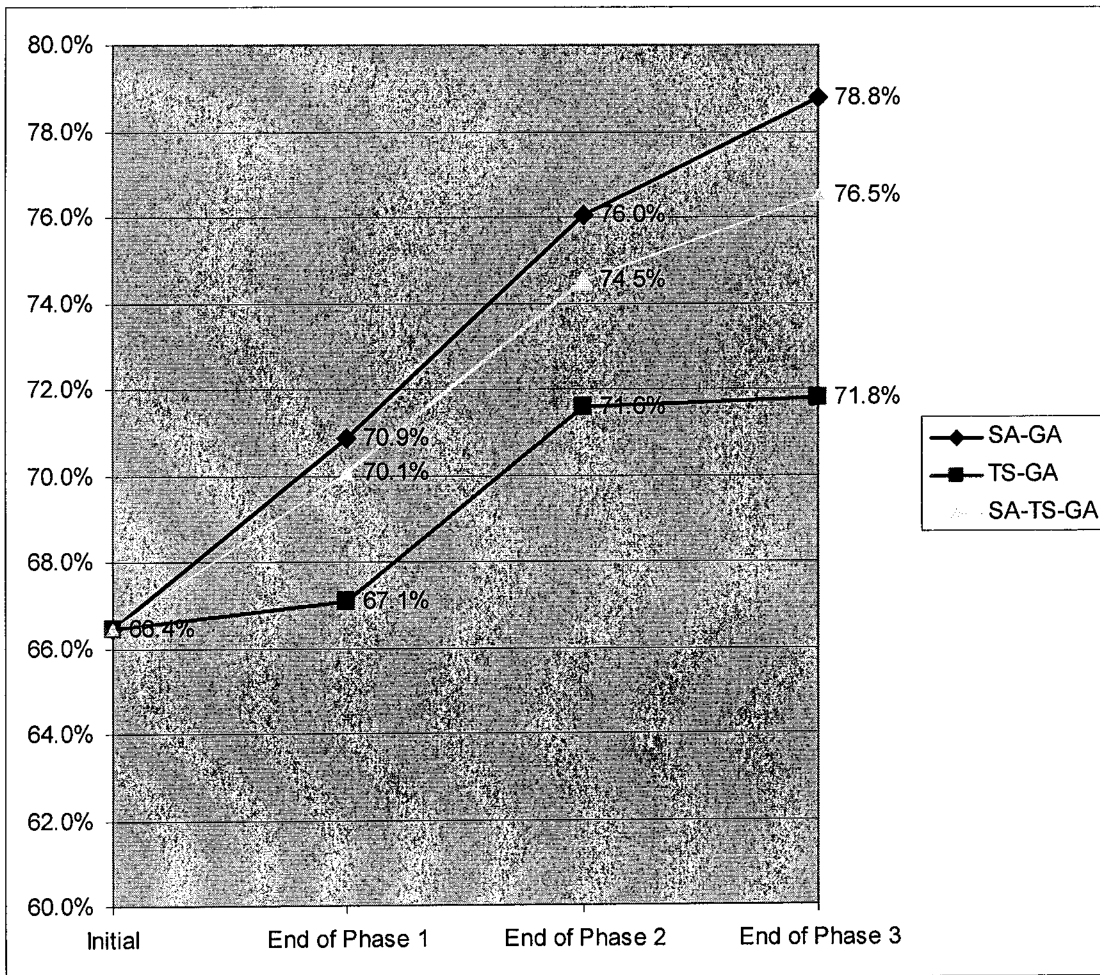


Figure 27: Plot showing the improvement in J-Index in Experiment 3 on STAB1 at the three phases of optimization for each hybrid heuristic.

CHAPTER 7: CONCLUSION AND FUTURE WORK

The core contribution of this thesis resides in the design, implementation and testing of three new hybrid heuristics to optimize existing software quality estimation models. The intrinsic originality of the heuristics is that they are hybrid both in nature and in algorithmic configuration. They constitute a hybrid approach as they are designed to work on two levels of optimization simultaneously: The optimization of a single rule set by adapting it to a dataset as well as the optimization of a set of rule sets by combining them and adapting them to a dataset. Moreover they combine different heuristic optimization techniques: The first hybrid heuristic, SA-GA, is a combination of Simulated Annealing and Genetic Algorithms. The second hybrid heuristic, TS-GA, combines Tabu Search with Genetic Algorithms. The third and last hybrid heuristic, SA-TS-GA, combines Tabu Search, Simulated Annealing and Genetic Algorithms. When tested on two different datasets describing OOP component stability, all three hybrid heuristics were able to achieve superior results when compared to C4.5. SA-GA showed to be the most performing one of the three. Furthermore, experimental results showed that the SA-GA outperformed, even if slightly, previous techniques that were tested on the same datasets. This slight improvement is not discouraging as it can be due to the fact that the datasets are noisy.

An interesting future work path would be to thoroughly test the hybrid heuristics on different datasets, not necessarily from the software quality estimation field (e.g. bioinformatics datasets). Another future work path would be to design and implement new hybrid heuristics that

could incorporate the skills of diverse optimization techniques such as ant colony and hill climbing.

References

Aarts, E. & Korst, J. (1989). *Simulated annealing and Boltzmann machines.* New York: Wiley.

Abreu, F. & Melo W. (1996). Evaluating the impact of object-oriented design on software quality. *Proceedings of the 3rd International Software Metrics Symposium,* 90-99.

Andersson, T. (1990). *A survey on software quality metrics.* Finland: Abo Akademi University.

Azar, D. (2004). *Using genetic algorithms to optimize software quality estimation models.* Unpublished doctoral dissertation, McGill University, Canada.

Azar, D., Precup, D., Bouktif, S., Kegl, B. & Sahraoui, H. (2002). Combining and adapting software quality predictive models by genetic algorithms. *Proceedings of the 17th IEEE International Conference on Automated Software Engineering,* 285-288.

Barker, J.A. & Henderson, E.A (1976). What is "liquid"? Understanding the states of matter. *Reviews of Modern Physics, 48,* 587-671.

Boswell, R. (1990). *Manual for NewID.* Glasgow: The Turing Institute.

Bouktif, S., Azar, D., Sahraoui, H., K´egl, B. & Precup, D. (2004). Improving rule set-based software quality prediction: A genetic

algorithm-based Approach. *Journal of Object Technology, 3(4)*, 227-241.

Chidamber, S. & Kemerer, C.F (1994). A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering, 20(6)*, 476-493.

Clark, P. & Niblett, T (1989). The CN2 induction algorihtm. *Machine Learning, 3*, 261-283.

Cormen, T.H, Leiserson, C. E., Rivest, R. L & Stein, C. (2001). *Introduction to algorithms* (2nd ed.). Massachusetts: MIT Press.

Curtis, B. (1980). Measurement and experimentation in software engineering. *Proceedings of the IEEE, 68*, 1144-1147.

Darwin, C. R. (1859). *On the origin of species by means of natural selection or the preservation of favored races in the struggle for life.* London: John Murray.

De Almeida, M.A. & Matwin, S. (1999). Machine learning methods for software quality model building. *International Symposium on Methodologies for Intelligent Systems*, 565-573.

De Jong, K. A. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* Unpublished doctoral dissertation, University of Michigan, Ann Arbor, Michigan.

Evett, M., Chien, P., Khoshgoftar, T. & Allen, E. (1998). GP-based software quality prediction. *Genetic Programming: Proceedings of the Third Annual Conference*, 60-65.

Fenton, N. & Neil, M. (1999). Software metrics and risk. *The Journal of Systems and Software, (47),* 149-157.

Galin, D. (2004). *Software quality assurance: From theory to implementation.* Harlow: Pearson.

Gao, K. & Khoshgoftaar, T. (2007). Count models for software quality estimation. *IEEE Transactions on Reliability, 56(2),* 212-222.

Gendreau, M. (2002). *An introduction to tabu search.* Montréal: Université de Montréal.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research, 13,* 533-549.

Glover, F. & Laguna, M. (2002). *Handbook of applied optimization.* New York: Oxford University Press.

Holland, J.H. (1975). *Adaptation in natural and artificial systems.* Michigan: University of Michigan Press.

Jones, W.D. & Khoshgoftaar, T.M. (1998). Using classification trees for software quality models: Lessons learned. *The 3rd IEEE International Symposium on High-Assurance Systems Engineering,* 82-89.

Kan, S.H. (2002). *Metrics and models in software quality engineering* (2nd ed.). Boston: Addison-Wesley.

Khoshgoftaar, T.M., Allen E.B., Halstead, R., Trio, G.P., & Flass, R.M. (1998). Using process history to predict sofware quality. *Computer, 31(4),* 66-72.

Kirkpatrick, S., Gelatt, C.D. & Vecchi, M.P. (1983). Optimization by simulated annealing. *Science, 220,* 671-680.

Metropolis, N., Rosenbluth, A., Rosenbluth, T., Teller, A. & Teller, E. (1953). Equation of state calculations by fast computing machines. *Journal of Chemical Physics, 21,* 1087- 1092.

Mish, F.C. (1993). *Merriam – Webster collegiate dictionary* (10th ed.). Massachusetts: Merriam-Webster.

Mitchell, T.M. (1997). *Machine learning.* New York: McGraw-Hill.

Pan, J. (1999). *Software reliability.* Pittsburgh: Carnegie Mellon University.

Pressman, R.S (1988). *Making software engineering happen, a guide for instituting the technology.* New Jersey: Prentice Hall.

Quinlan, J.R. (1990). Learning logical definitions from relations. *Machine Learning, 5(3),* 239-266.

Quinlan, J.R. (1993). *C4.5: Programs for machine learning.* San Francisco: Morgan Kaufmann.

Reeves, C.R. & Rowe, J.E. (2003). *Genetic algorithms – Principles and perspectives.* Boston: Kluwer Academic Publishers.

So, S.S., Cha, S.D. & Kwon, Y.R. (2001). Empirical evaluation of a fuzzy logic-based software quality prediction model. *Fuzzy Sets and Systems, 127*, 199-208.

Sommerville, I. (2004). *Software Engineering* (7th ed.). London: Addison-Wesley.