GENETIC TUNING

TUNING THE MATCHING APPLICATION ELISE USING MACHINE LEARNING TECHNIQUES

Suzanne van Gelder

Artificial Intelligence (Cognitive Science and Engineering)
University of Groningen

Cap Gemini Ernst & Young, Bolesian
Utrecht

GENETIC TUNING

TUNING THE MATCHING APPLICATION ELISE USING MACHINE LEARNING TECHNIQUES

by
Suzanne van Gelder
s0990396
s.van.gelder@tricat.nl

1 September 2001 - 28 February 2002

Artificial Intelligence (Cognitive Science and Engineering)
University of Groningen

Cap Gemini Ernst & Young, Bolesian
Utrecht

Supervisors
Rineke Verbrugge (AI/TCW)
Toine van Moergastel (CGE&Y, Bolesian)
Sebastiaan Ubink (CGE&Y, Bolesian)

CONTENTS

List of Tables

List of Figures

List of Equations

Acknowledgements

I want to thank Rineke Verbrugge (AI/TCW) for her supervision, Sebastiaan Ubink (CGE&Y, Bolesian) for his technical guidance and feedback, Toine van Moergastel (CGE&Y, Bolesian) for his guidance, Geert Krekelberg (CGE&Y, Bolesian) and Dick van Soest (CGE&Y, Bolesian) for their assistance, Peter Teijgeman (CGE&Y, Bolesian), Jacques Dunselman (CGE&Y, Bolesian), and Bianca Willems (CGE&Y, Bolesian) for their arrangements and Peter Went (WCC), Freek Geerdink (WCC) and Mark Wegman (WCC) for their technical support and feedback.

Further I want to thank Seth Kingma for his mental support at home and his hard working to keep TriCAT *i*Consulting running without me, together with Jan Curganov. I also want to thank Hannie Eberhardt for her mental support, Cynthia van Weeren (CGE&Y, Bolesian and AI/TCW) for her traveling company and Thea Jongerius for a sleeping-place in Utrecht during my internship.

Abstract

Bolesian (a part of Cap Gemini Ernst & Young, specialized in knowledge-based solutions) develops matching applications for the vacancy and resume domain. Those matching applications are often used by HRM-departments (Human Resource Management) of companies or by temping agencies. Those matching applications assist them in finding the right employees for a given vacancy. On the other hand, employees can match their resumes against the available job openings.

A matching algorithm compares demand to supply to calculate how close the supply matches the demand. Those match scores are ordered and the company (or employee) receives a list with the highest scores that probably meet the demands of a specific vacancy (or resume). The candidate profile (or job) with the highest match score is likely to be the best suitable profile (or job) to the job (or resume).

One of the most difficult aspects of matching is tuning the application. Tuning is the balancing of the parameters of the match criteria so that the match results will appear in the *right* order, and the *good* resumes (or vacancies) score higher than a certain threshold and the *bad* ones score lower. This order differs from user to user (in our case HRM departments and temping agencies). One user puts a lot of emphasis on working experience, while another user values skills and education more highly.

The tuning of the parameters is a manual process. It can cost days or weeks to set all the parameters correctly. Given the great number of companies that use this kind of applications, manual tuning is not really attractive. It is clear that an automated tuning process can save a lot of time, so the idea was born to use machine learning techniques to learn the correct parameter setting.

The goal of this project was to determine whether or not machine learning techniques can be of use in tuning those parameters automatically. If so, which machine learning algorithms are appropriate and under what conditions can they be used. Therefore, it is investigated what can be learned by an algorithm and what must be defined within the domain.

I have implemented a genetic algorithm and tested it with training data of an existing project. It turned out that a genetic algorithm is appropriate to tune the parameters automatically. The test results showed that the algorithms converges towards an optimal solution that approximates the target match scores of the existing project closely.

Keywords: matching, (automatic) tuning, machine learning, genetic algorithm.

1. Introduction

About a year ago I was working on the last courses of my study. It was time to think of a thesis subject to graduate on. There were several requirements my final project had to meet. It had to be an internship at a company that was not located in Groningen. The project had to be a mix of theoretical study and practical assignment. Also it had to be related to the subject knowledge technology. At the same time the company Bolesian was looking for a trainee. Bolesian was not located in Groningen and offered a vacancy of an internship for learning algorithms related to knowledge technology.

This resulted in a "match". The outcome is this final thesis of my study Artificial Intelligence in Groningen (formerly Cognitive Science and Engineering), carried out on the part of Bolesian in Utrecht that is a service practice of Cap Gemini Ernst & Young.

Bolesian is specialized in knowledge-based solutions. This includes several services, like expert systems, matching and planning and scheduling. Matching forms the subject of this thesis.

Bolesian develops matching applications for the vacancy and resume domain. Those matching applications are often used by HRM-departments (Human Resource Management) of companies or by temping agencies. Those matching applications assist them in finding the right employees for a given vacancy. On the other hand, employees can match their resumes against the available job openings. A matching algorithm compares demand to supply to calculate how close the supply matches the demand. Those match scores are ordered and the company (or employee) receives a list with the highest scores that probably meet the demands of a specific vacancy (or resume).

The total match score between a resume and a vacancy is calculated using the weighted average of the scores of all properties involved in a match. The candidate profile (or job) with the highest match score will probably be the best suitable profile (or job). Properties used in the vacancy and resume matching domain are for instance position, salary, traveling distance, experience, education and skills (like language or computer skills).

A precise and time-consuming aspect of developing a matching application is tuning the application. Tuning is the balancing of the parameters of the match properties so that the match results will appear in the "right" order, and the "good" resumes (or vacancies) score higher than a certain threshold and the "bad" ones score lower. This order differs from client to client. One client puts a lot of emphasis on working experience, while another client values skills and education more highly. Therefore, it is necessary to tune the parameters in such a way, that the client gets the match results ordered and divided in "good" and "bad" results in an acceptable way.

The tuning of the parameters is a manual process. It can cost days or weeks to set all the parameters correctly. Given the great number of companies that use this kind of applications, manual tuning is no attractive option. It is clear that an automated tuning process can save a lot of time.

The idea was born that machine learning techniques should be able to learn the correct parameter setting based on training cases that are created and valued by the client. It must be investigated which learning algorithm is the most appropriate and can tune the parameters automatically. An existing matching project of Bolesian should be used to test the algorithm.

Definition

In matching applications match scores are calculated between demand and supply. The match scores are calculated using different parameter settings for different criteria. Those parameters are tuned by hand.

The goal is to determine whether or not machine learning techniques can be of use in tuning those parameters automatically. If so, which machine learning algorithms are appropriate and under what conditions can they be used? Therefore it must be investigated what can be learned by an algorithm and what must be defined within the domain. The most appropriate algorithm must be implemented and tested with training data.

The structure of this Master's thesis will be as follows. Chapter 2 will introduce the matching application Elise and directions to tune this application by hand. In chapter 3 the requirements, basic assumptions and limitations of the learning algorithm are given. Chapter 4 will discuss three different machine learning techniques. The most promising algorithm is chosen and will be used as automatic tuning algorithm. In chapter 5 the design of the selected algorithm is specified. In chapter 6 the implementation phase is evaluated. Chapter 7 outlines the experiments and its test results and the thesis will be concluded with a discussion in chapter 8.

In this paper "AI/TCW" will refer to the study Artificial Intelligence and "Bolesian" to the service practice Bolesian of Cap Gemini Ernst & Young.

Suzanne van Gelder

Utrecht, 28 February 2002

2. Matching Domain

In this chapter the matching domain is described. First the matching application Elise is introduced. After that, the DataDictionary of Elise is discussed. In this dictionary the domain of a specific matching project is declared. Finally, the tuning process is described. Tuning is the balancing of weights in the DataDictionary, so Elise will calculate the correct match scores, which are the match scores that are desired by the client.

For the discussion about Elise and the DataDictionary documentation of the Elise software (WCC, 2001) is used as source. The information of the tuning process has been gathered during an interview with tuning expert Sebastiaan Ubink.

2.1. Elise 4.0

Elise is a matching tool that calculates match scores between a demanded and an offered side. It is often used for vacancy matching, where vacancies are matched against resumes or where resumes are matched against vacancies. Other possible matching domains are 'hospital beds' with 'free beds in a hospital', and 'new patients without a bed' or 'cars' with 'cars for sale' and 'profiles of searched cars'. All those matches are two-sided, they have something to offer and they demand something. Within the vacancy domain vacancies offer for instance a position and a salary and they demand an education and experience. On the other hand, resumes offer education and experience and demand a position and the corresponding salary instead. This is shown in figure 2.1.

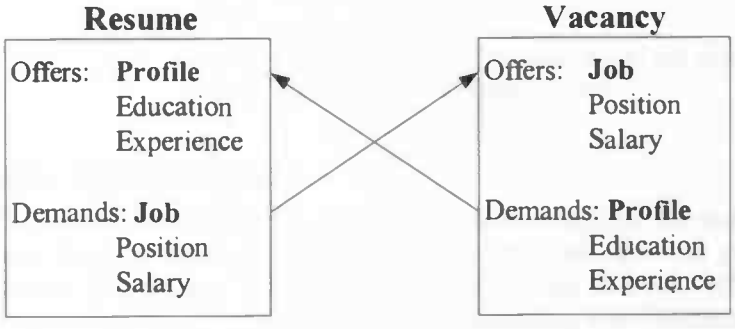


Figure 2.1. *Matching*

In Elise, the vacancies and resumes are called deals. As mentioned above, each deal consists of a demanded side and an offered side. Those are called the products. For the vacancy deal those are the profile (demanded) and the job (offered). For a resume deal it's the other way around. The criteria like position, salary, education and experience are the properties of a deal. The match between a vacancy and resume will be calculated based on those properties and the related values of a specific vacancy and resume. For example, values can be 'high school' as education or '50,000 EURO' as salary. All the definitions of used properties and possible values are defined in the DataDictionary, which is described in the next section.

To calculate a match score, Elise uses fuzzy searching instead of hard searching. Using the last, a match must be a perfect match; every offered and demanded property must match 100%, otherwise it's a nomatch. With fuzzy searching a vacancy and a resume can also match for 80% or 50%. This occurs when not all the demanded properties match the offered ones perfectly. For example when the vacancy demands someone who speaks fluently Spanish, French and German

but someone who is offering his or her skills only speaks fluently Spanish and French but no German. This person can still be interesting for the company that is searching an employee, so Elise must offer the resume of this person to the company. Another example is a vacancy that is demanding a commercial employee and a candidate who is searching a position as purchasing agent. The job doesn't match perfectly, but probably is appropriate, so it should match for about 85%. This is defined by a matrix (see subsection 2.2.1).

To calculate the total match score the formula in equation 2.2 is used. First all properties of both demanded sides are matched against the corresponding offered side to calculate the product scores using equation 2.1. This results in two match percentages, one job match percentage and one profile percentage. The weighted average is calculated to obtain the total match score.

equation 2.1. *product_score* (WCC, 2001)

$$product\_score = \frac{\sum_{demanded\ properties} actual\ score}{\sum_{demanded\ properties} maximum\ score} \cdot 100\%$$

with the *actual score* and the *maximum score* dependent on the definitions in the DataDictionary (see next section)

equation 2.2. *match_score* (WCC, 2001)

$$match\_score = \frac{\%_1 \cdot A + \%_2 \cdot B}{A + B}$$

with the product score of the job·A and the product score of the profile·B, mostly A is equal to two and B equal to one

2.2. DataDictionary

In the DataDictionary the matching domain is defined. Of each domain the matching properties and their parameter settings, hierarchical relations between properties and possible values and match percentages between different values are defined. Below a description is given of the topics that are of importance for understanding the matching domain and the topics that are relevant to this internship. Those are PRODUCT.DIC, master-slave relations and ranges and gliding scales.

2.2.1. PRODUCT.DIC

In the PRODUCT.DIC file the offered side is defined. This means there are two PRODUCT.DIC files in one matching project, because there are two offered sides. The PRODUCT.DIC defines the match properties like position and salary and of each property the parameter settings are defined. An example is given below.

```
- POSITION, DOM(LIS), WEI(0, 0, NEVER, 5000), MIN(1, 0, 1, 0),\
            MAX(1, 1, 1, 1000), TYP( ,OR), VAL(position), MAT(position)
      @UK,  "Position"
```

In this example, "POSITION" is the property. DOM, WEI, MIN, MAX, TYP, VAL and MAT are parameters with the related value(s) between brackets. On the last line @UK is related to the language, in this case English and "Position" is the property name used by the interface. The parameters in the definition of this property are used most common and give a good picture of the

matching domain. Therefore, only those parameters are discussed below.

DOMain(LISt | FREe | NUMeric | DATe)

The DOMain parameter defines the type of value that must be used by the property. This can be a fixed list, free text, a numeric value or a date.

WEIght(<NOOBJECT>, <NOVALUE>, <NOMATCH>, <MATCH>)

The WEIght parameter specifies the relative importance of a property. It consists of four attributes that represent four different situations. The situation NOOBJECT occurs when the demanded property does not exist on the offered side. The situation NOVALUE occurs when the property exists, but has no value. The NOMATCH situation occurs when the demanded property exists, has a value, but doesn't match the demanded value and the MATCH situation occurs when the demanded property exists and has a value that matches the demanded value.

The possible values of the four WEIght attributes are NEVER, ALWAYS and all integers between –32,000 and +32,000. When a property matches with a NEVER, the match score of the related side will be 0% and when a property matches with an ALWAYS, the match score of the related side will be 100%, without evaluating the other properties. When the property matches with a numeric value, the match score will be the percentage of the numeric value related to the highest occurring WEIght value of that property. For example when the MATCH WEIght is 4000 (and is the highest WEIght value) and the match percentage is 75%, the match score of the property will be 3000.

MINinstances(<inst off> [, <inst dem> [, <val off> [, <val dem>]]])

With the parameter MINinstances the minimum number of instances of a property and the minimum number of values for each instance is defined. This parameter contains four attributes, namely <inst off>, <inst dem>, <val off> and <val dem>, which represents respectively the minimum number of offered instances, demanded instances, offered values and demanded values.

MAXinstances(<inst off> [, <inst dem> [, <val off> [, <val dem>]]])

With the parameter MAXinstances the maximum number of instances of a property and the maximum number of values for each instance is defined. This parameter contains four attributes, namely <inst off>, <inst dem>, <val off> and <val dem>, which represents respectively the maximum number of offered instances, demanded instances, offered values and demanded values. Both MINinstances and MAXinstances are related to the TYPe parameter, which is discussed next.

TYPe([<RepeatingGroupType>] [, <MultiValueType>])

The TYPe parameter has two attributes. The RepeatingGroupType and the MultiValueType. The RepeatingGroupType attribute is used for a property that can be repeated in one deal. The MultiValueType attribute is related to the occurrence of multiple values of one property.

Of the MINinstances and MAXinstances the <inst off> and the <inst dem> attributes are related to the RepeatingGroupType and the <val off> and <val dem> attributes are related to the MultiValueType.

RepeatingGroupType

The possible values of the RepeatingGroupType are EXClusive, REUse and OR. When EXClusive is set, an offered property can be used only once in one match and the score of all properties is summed. When REUse is set, an offered property can be used more than once and the score of all properties is summed.

When OR is set, an offered property can be used more than once, but only the highest property match result is used. For example when, property 1 is demanded with value x and WEI(0, 0, 0, 100) and property 2 is demanded with value y and WEI(0, 0, 0, 300) and x and y are offered, the match score is (300 / 300) * 100 = 100%. For EXClusive and REUse the score would be ((100 + 300) / (100 + 300)) * 100 = 100%.

MultiValueType

The possible values of the MultiValueType are OR, AND and INTersection. When the value OR is set, at least one of the properties must match to have a match, when no properties match it's a nomatch. When the value AND is set, there is only a match when all properties match, otherwise it's a nomatch. When the value INTersection is set, the more properties match, the better the match will score. For example when five values are demanded and one of them is offered, the score is 20%, when three are offered, the score is 60%, and when five are offered it's a 100% match.

VALues(<filename> | <from value>, <till value>)

The VALue parameter specifies the possible values for the related property. With DOMain(LISt) it refers to a file with declared objects (see table 2.1.a), with DOMain(NUMeric) or DOMain(DATe) it refers to a from value and a till value (numeric or date).

MATrix(<filename>)

The MATrix parameter is only valid for properties with LISt defined for the DOMain parameter. The MATrix parameter refers to a file with a matrix (see table 2.1.b). This matrix defines the match score percentages between objects in the list. The first number stands for the demanded object value, the second number stands for the offered object value and the last number represents the match score between the two objects.
The matrix is used for offered and demanded objects that show resemblances, but that are not identical. For example, when computer science is the demanded education and artificial intelligence is the offered education. This doesn't result in a perfect match, but when it is defined as a 75% match in the matrix, the match percentage of this property is 75%.

Position.uk

```
1, "Secretary"
2, "Assistance"
3, "Project Manager"
4, "Director"
```

Table 2.1.a. *Values*

Position.mtx

```
DEFAULT

# Demanded, Offered, Percentage
1, 2, 75
2, 1, 60
3, 4, 75
4, 3, 60
```

when DEFAULT is defined, all missing match combinations (except perfect matches) are valued with 0%, which is a nomatch

Table 2.1.b. *Matrix*

2.2.2. Master-Slave Relations

A master-slave relation is a special kind of hierarchical relation between properties. This relation is registered in the PRODUCT.DIC file and influences the match behavior. An example of the master-slave relation is the following. There are two properties; one is "course" and the other "certificate". In this example "course" is the master and "certificate" is the slave. This means that the occurrence of a "certificate" is only taken into account when it is related to the corresponding "course", so a match on certificate without a match on course will result in a 0% score. When both properties are matches, the score will obviously be 100%. When only "course" is a match, the score will be calculated as usual.

2.2.3. Ranges and Gliding Scales

There are different ways to determine the match percentage between a demanded and an offered property. One is using a matrix, which is already mentioned. In the matrix the match percentages between two objects of a list are defined. Besides the matrix there are ranges and gliding scales. Those define the match percentages between two properties of the numeric value or date type. When a range is defined for the demanding deal, the value of the offering deal must lie within the range to be a match (100%), otherwise it will be a nomatch (0%). When the range has only a minimum and a maximum value, the borders are sharp. When an absolute minimum and an absolute maximum are also defined, gliding scales are created (see figure 2.2). With gliding scales the match percentage can be 0% till 100%.

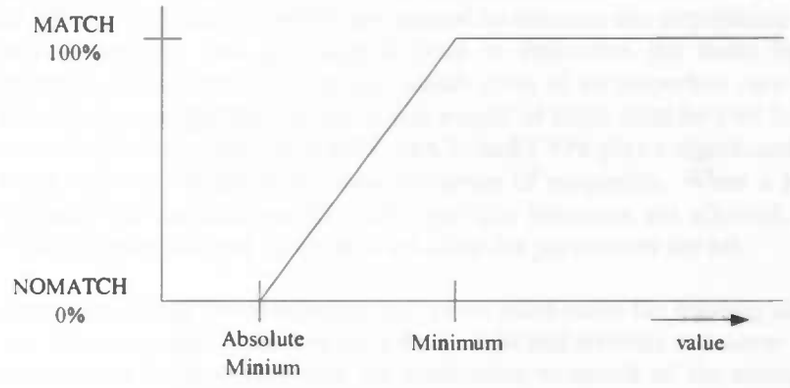


Figure 2.2. *Gliding Scale*

2.3. Tuning

Tuning is the process of balancing the parameters of the different properties defined in the PRODUCT.DIC files, like WEIght, MINinstances, MAXinstances and TYPe. This is a manual and time consuming process, because it is not always easy to determine the parameters that must be changed, and what this change must be. A number of test cases are used to evaluate the performance of Elise. With the match scores as results, the errors are traced and rectified. Every time an application is developed, tuning is a part of the developing process. Due to the big differences between the requirements of each client, the applications differ a lot, so projects cannot be reused. Below the design of the application, the initialization and the tuning process are described.

Looking at the vacancy matching domain again, a few examples of differences between requirements of clients are given. These examples will give a good impression of the possible variety between matching applications. First clients can value properties different. Some clients value working experience highly while another client values the skills (like speaking foreign languages) more highly. Also the used matching properties can differ. It can be useful for a client to define the traveling distance as a property when the application must be developed for a temping agency, but when a company wants to use the application intern on one location, the property is unnecessary. Further the multi instances can differ. One client might think it is useful when a user can fill in all his or her working experiences, while another thinks only the last two experiences are of importance. At last the hierarchy of the properties is not static. Properties like education and diplomas or function and position can or cannot be related to each other.

All those differences mean that a new matching domain must be developed for every project to meet the wishes of each client. Therefore it must be determined which information is used to match vacancies with resumes and what kinds of properties play a role in the matching process. Knowledge acquisition (see glossary) is used to extract all the needed information from the client. The gathered information is the source for a list with all the properties that must be used in the domain. The list is divided into a resume demanded side and a vacancy demanded side and related properties are grouped together. It is also determined what kind of data is needed per property, like an integer for traveling distance and salary and a list for education and position. Based on this detailed list the functional design is made.

Once the structure of the DataDictionary is defined and implemented it is time to set the

parameters in the PRODUCT.DIC files. This initialization is based on experience of the client. In an interview all the properties are treated to discover the importance of each property. Often the cart-sorting tool (see glossary) is used to determine the order between the properties. For example, the influence on the total match score of all properties may be the same, but skills are a little bit more important, so the match weight of skills must be a bit higher. Not only WEIghts are evaluated in this stage. Also MIN, MAX, and TYPe play a significant role. These are determined using information about the total influence of properties. When a property must have a lot of influence on the total match score, multiple instances are allowed, otherwise multiple values. When all relations and influences are clear the parameters are set.

Before the tuning process begins, the client must make the training cases. These cases have to be real-life cases, which contain the average jobs and resumes and cover all the used properties. This is important because you want the application to match all the normal cases right. The client is instructed to make those cases. Each case set contains one vacancy, and several resumes, or one resume, and several vacancies. The cases must be bounded to some requirements, which are defined below.

- Each match within the case set must contain the following information: the target match score (or an approximation) and the fact match or nomatch. When it is hard to determine the scoring percentages, it can be simplified with case sort matching: the order between the matches will appear. Due to this order, approximations of the match percentages can be derived.
- All the different properties must be present in the collection of cases (for example traveling distance, skills, and experience) so all aspects will be taken into account while tuning.
- Examples of matches are always required. Examples of nomatches are only required when there are differences between deals that don't score a NEVER. For example the offered function matches for 50% with the demanded function (which is not a NEVER), this can be a match when the skills also match, but a nomatch when skills don't match. When all deals that don't score a NEVER must match, nomatch cases can be left out.
- Inconsistencies between cases are not allowed. When cases are equal (the match percentages of all properties are equal), their target match scores must be equal.
- For all the cases the threshold must be the same, so all match scores above the threshold are considered as matches and the scores below the threshold are considered as nomatches.

The number of needed cases depends on the complexity of the domain. When a lot of properties must be tuned, more cases are needed than when only five properties must be tuned. This is required, because there must be enough variation between the cases, as mentioned above.

When the client has provided all the correct cases, the cases are imported into Elise and the tuning process begins. The match scores are calculated and of each case the actual result is compared to the target match score. Based on the differences between the match score, the target score and related information from the cases, the parameters in the PRODUCT.DIC files are modified. However, not only those parameters can be incorrect. Sometimes there are inconsistencies in the design, like a hierarchical relation or a matrix. Those problems are difficult, but they aren't taken into account below, because they are considered to be correct when tuning with a learning algorithm.

During each tuning cycle, feedback from the performance of Elise concerning the cases helps to find the most obvious error. Comparing the expected results with the actual results, it can be detected where matches go wrong, based on the knowledge of the tuning expert. He or she knows why some deals must match or not. For example the influence of the property traveling distance

is too high, which results in too high match scores for cases with a matching traveling distance. In such a case, the MATCH WEIght of corresponding property is lowered. When the most obvious error is detected, only the parameter of the related property is changed, otherwise the effects of this change aren't clear.

For each parameter attribute there are reasons to set or modify it to a specific value. Those are given below illustrated with some examples.

NOOBJECT — The value of NOOBJECT is often the same as NOMATCH or it is zero. In the first case NOOBJECT is treated as a nomatch. In the second case zero is used because NOOBJECT doesn't play a role, because it will never occur (due to the design).

NOVALUE — NOVALUE can be used to score less than 100%. This can be done when a property is less important than the other properties. The value is set higher than the MATCH WEIght. However, in those situations NOVALUE is set to zero and the MATCH WEIght is lowered.

NOMATCH — Most of the time, NOMATCH is set to zero or NEVER. NEVER is used when the property is required, zero is used in the other occasions. Besides that, negative weights are sometimes used as penalties. When a property has a non-qualifying influence (which means it is not absolutely required, so it can be compensated) a penalty is given when the property scores a NOMATCH, so the total result still can be a match.

MATCH — Most of the time, positive values are used as match values, so credits are given in the case of a match. The height of the numeric value is dependent of the importance of the property.
However, a negative value is also possible. This occurs in case of disapproval. For example when an employee doesn't want to work in an office where people smoke, the property must have a negative influence in the case of a match.

NEVER — The decision to use a NEVER is based on the feeling of the user: "must the deal be offered or not when there is a mismatch on the property", most of the time it can be estimated very well if a NEVER must be used. NEVER is often used when a matrix or gliding scale is available for the specific property. When the score is 0% the object is too different, so the deal isn't worth offering.
"Required" is a special match behavior, which is often used. The value NEVER is set on the NOMATCH when the user requires a demanded property.
NEVER on the MACTH is used when a deal must not contain a certain property.

ALWAYS — ALWAYS is not used very often, but a typical example is the following: an employer prefers a certain employee due to past experience, so a match with that particular employee will always be a good match.

MultiInstances/MultiValues — The first choice to be made is whether or not a property or the value of a property is allowed more than once. When the answer is yes, the proportion of the property related to the total match score must be

determined. When the influence must be high, MultiInstances are chosen, otherwise MultiValues. Often this becomes clear with the provided cases.

TYPe(EXClusive | REUse | OR) The value REUse is most common.

TYPe(, OR | AND | INTersection) The value OR and INTersection are most common. The choice depends on the current and wanted matching behavior of the client.

It should be avoided to use NEVER and ALWAYS simultaneously in one PRODUCT.DIC. Besides the mentioned example concerning the particular employee that must match ALWAYS, it is of no use defining them both. When a NEVER is calculated on a result, the product score will directly be set to 0% (for an ALWAYS this is 100%). It is neglected when another property results in an ALWAYS (NEVER), because the score of 0% (100%) is already ascribed to the product. This means that the weight of the first property is qualifying for the calculated match percentage.

After the process, when all matches are matches, all nomatches are nomatches, the actual match scores lie within an acceptable range from the target scores and the order between the matches is correct, all parameters are tuned and the design and matrices are correct, so the tuning process is finished.

Taking all remarks into consideration, tuning is a difficult and time-consuming process. It would save a lot of time when a machine learning algorithm could tune the DataDictionary, even if it is just a small part like the PRODUCT.DIC file. For example a few difficult parameters like the WEIght attributes that could be tuned by an algorithm. In the next chapter the requirements a learning algorithm should meet, are described.

3. Requirements, Limitations and Basic Assumptions

In this chapter the requirements, limitations and basic assumptions are defined to demarcate the project. The section on requirements defines the parameters that must be learned by the algorithm and it defines the factors that make automatic tuning interesting for Bolesian and her clients. The algorithm must fulfill those requirements to be successful. Problems that don't have to be solved by the learning algorithm are described in the section on limitations. In the last section the basic assumptions are defined. Those must be met to learn a solution.

3.1. Requirements

This section describes the requirements for the tuning problem that must be fulfilled. The learning algorithm will be selected based on the ability to fulfill those requirements so the algorithm can be successful and will be interesting to Bolesian.

The learning algorithm must be able to learn an optimal parameter set. The parameters that must be learned are listed below together with related requirements.

- For all the properties in the PRODUCT.DIC files the WEIghts must be learned. This includes the noobject, novalue, nomatch, and match weights. Each weight can be a NEVER, an ALWAYS, or an integer between –32.000 and +32.000.
- TYPe must be learned for the properties which are defined as multi instances (TYP(EXClusive/REUse/OR , _)), or as multi values (TYP(_ , OR/AND/ INTersection)), or both (TYP(EXClusive/REUse/OR , OR/AND/ INTersection)).
- The threshold for the boundary between a match and a nomatch must be learned. A threshold of 50 means that a total score below 50 percent is interpreted as a nomatch, and a score of 50 percent or higher as a match. It depends on the cases if the threshold must be learned (see section 5.2).
- The user must have the possibility to designate parameters as "don't have to be learned by the algorithm". The algorithm must recognize those parameters as fixed and is not allowed to change them. This is particularly useful for parameters that are known to be of type "required" and therefore must score a NEVER on NOMATCH.
- The algorithm must stop when the termination criterion is met. This means that most of the matches are identified as matches and nomatches as nomatches. Further, the percentage of most matches fall in the range "target score minus or plus *x* %", or the order between the matches within the cases must be right or both range and order must be met (see section 5.2).

Besides those requirements, other general requirements are also of importance to Bolesian. Those are listed below.

- The automatically learned parameter set must perform at least as well as the parameter set that is tuned manually.
- The learning algorithm must be able to tune the parameters within a reasonable limit of time. Compared to tuning by hand, automatic tuning must be faster and it must not take more working hours to prepare the trainings and test cases.
- The algorithm must be able to tune different domains. This means that it must be applicable on all possible domains comparable to the vacancy domain and that the matching subject or the number of properties are of no importance.
- The amount of training and test cases needed by the algorithm must not be too large.

- It must be easy to change the settings of the algorithm. The settings of the algorithm are the parameters that steer the learning algorithm.
- It must be easy to extend the learning possibilities of the algorithm, for example the possibility of learning extra parameters.
- The chance on success in real life (the algorithm meets all the requirements) must be high.

3.2. Limitations

Besides the requirements it is decided that there will be one limitation. This limitation concerns dynamic matching applications. When dynamic matching applications are used the weights are determined in the code depending on which properties are known and which are not. So they are not extracted from the PRODUCT.DIC files. Tuning this type of applications automatically will be very difficult. So dynamic matching applications don't have to be learned by the algorithm.

3.3. Basic Assumptions

Finally, the basic assumptions are discussed. The learning algorithm can only be used when those assumptions are met, otherwise it will not converge to an optimal solution. So when all the requirements are met and the algorithm stops learning, but it doesn't return an optimal parameter set, the basic assumptions are not met. The algorithm will not detect the errors, so the design must be checked and corrected.

The assumptions are listed below.

- The design of the domain must be correct.
- All the defined matrices in the DataDictionary must be correct.
- All the defined gliding scales must be correct.
- The algorithm must have enough training data to learn from and enough test data to test with. The amount of data will be discussed in section 5.2.
- The training and test cases must be correct and consistent, for that reason the threshold used by the cases must be the same for all the cases.
- The client must be able to define cases that meet the requirements (see section 5.2).

4. Learning Algorithms

A learning algorithm will be used to tune the parameters of Elise. This algorithm will use a number of cases (like the cases used when tuning by hand) to learn from and another set of cases with which the learned parameters can be tested. To learn well, the algorithm has to fulfill several requirements to converge to an optimal parameter set. These requirements will be discussed below.

To tune the parameters for Elise some learning algorithms are appropriate and some are not. The requirements the algorithm must fulfill are mentioned in chapter 3. Summarized, the algorithm must be able to learn discrete values (NEVER and ALWAYS, TYPe(EXClusive), TYPe(REUse), TYPe(OR), or TYPe(, AND), TYPe(, OR), and TYPe(, INTersection)), and continuous weights (an integer between –32.000 and +32.000 and the threshold). It must also be able to fix one or more values. Furthermore, the algorithm must be easy to apply to other domains (high generalizability), it must be easy to extend the learning possibilities of the algorithm and the number of needed training and test cases must not be too great (the client can't deliver ten thousand different cases). This latter aspect doesn't only depend on the learning algorithm: it also depends on the complexity of the domain. When the application uses a lot of properties, more cases are needed than when only a few properties are used. Besides that, the algorithm must be robust, it must be able to change the settings of the algorithm easily, it must give a time profit and the chance on success in real life must be high.

In this chapter, three algorithms are selected. Those algorithms are genetic algorithms, decision tree learning and knowledge-based learning. They meet most of the conditions mentioned above. The only method that doesn't meet all conditions is decision tree learning, which is a method for learning discrete-valued functions. When some changes are made, however, it can handle continuous values as well. Most other algorithms, like k-nearest neighbor (classification) or learning sets of rules, deal with other learning problems instead of learning weights. Neural networks do change weights in order to learn a target, but the weights themselves are not the targets, mostly a classification task is. So neural networks are not appropriate to learn both discrete and continuous values.

Genetic algorithms and decision trees are both forms of inductive learning: they use the training data to generalize and learn the optimal parameter set. Knowledge-based learning is based on a combination of analytical learning and inductive learning and uses prior knowledge and training data to learn from. All are forms of supervised learning because they receive feedback from the examples, like "this is a good match" or "this isn't a good match". Each method is described below. First, the basic idea, the method and the main characteristics are described. Then the method is applied to the tuning problem and the pros and cons are mentioned and possible problems are dealt with by presenting a solution. Knowledge-based learning will be discussed first because it contains a method that is also relevant for genetic algorithms and decision tree learning.

At the end of the chapter a reasoned comparison will be made between the algorithms to select the most appropriate for the problem of this project. For this balance the pros and cons of each algorithm are given. The chosen algorithm will be further discussed in chapter 5 and will be implemented to test its performance.

4.1. Knowledge-based Learning

Analytical learning uses prior knowledge and deductive reasoning to learn a problem by

augmenting information from the training examples. When using this knowledge, more correct generalizations can be produced from fewer examples than using no prior knowledge, but only when the knowledge is approximately correct and complete. The domain specific knowledge helps to analyze the features of the training examples, so the algorithm only take the relevant features into account during learning. This way the complexity of the hypothesis space is reduced, and search is simplified. Often analytical learning is combined with inductive learning because no complete and correct knowledge is available (T.M. Mitchell, 1997).

Normally the prior knowledge is offered to the algorithm as Horn clauses (see glossary). Look for example at the domain theory of a canary: $canary(x) \leftarrow bird(x) \wedge yellow(x)$ and $bird(x) \leftarrow wings(x) \wedge feathers(x) \wedge able\text{-}to\text{-}fly(x)$. Those clauses define when some animal is a canary (which is the target concept). As training examples only positive ones are given, like *yellow(animal_1)*, *wings(animal_1)*, and *small(animal_1)*. The output hypothesis must be consistent with the domain theory and the training examples, so the output hypothesis must at least "contain" $canary(x) \leftarrow small(x) \wedge yellow(x) \wedge wings(x) \wedge feathers(x) \wedge able\text{-}to\text{-}fly(x)$.

It is obvious that using Horn clauses will not be the way to tune the parameter set. There is no Horn clause to tune a parameter and developing Horn clauses to tune a parameter set is not possible because rules like $makeNomatchNever(x,y) \leftarrow 95Percent(z_1) \wedge negativeValue(x,y)$ and $makeMatch1000(x,y) \leftarrow 60Percent(z_1) \wedge tooSmallValue(x,y) \wedge 65Percent(z_1)$ can't be defined for the simple reason that tuning can't be described in terms of attributes and if-then rules. Therefore, this kind of prior-knowledge is not available. However, there is other knowledge that can be used, namely knowledge from tuning the parameters by hand and a lot of experience. This knowledge is discussed in the tuning section 2.3. A lot of this knowledge can be described by rules and constraints and there is a lot of implicit knowledge. For example, the knowledge used to determine what is the "most striking error at this moment and which parameter has the biggest influence on that error, so that that parameter can be changed first". Subsection 4.2.2. discusses the algorithm and possible rules for the implicit knowledge are given also.

When is analytical learning the best choice and when inductive learning? When available knowledge of the domain theory is complete and correct, analytical learning is the best way to learn (see above), but when there is lack of good domain-specific knowledge (it's incomplete) a combination of analytical learning and inductive learning can be used. When the knowledge is incorrect or it cannot be provided, pure inductive learning is the best method (T.M. Mitchell, 1997).

There are several approaches that use prior knowledge and inductive methods in combination to search through the hypothesis space. Only one of them is appropriate to the tuning problem, namely: deriving an initial hypothesis. This method is described in the next section.

4.1.1. Deriving an Initial Hypothesis

Before inductive learning starts, an initial hypothesis is derived. This hypothesis is defined using prior knowledge. The algorithm is put into the right direction with a derived initial hypothesis rather than with a randomly generated hypothesis, because the parameters are roughly tuned at the start. Therefore the algorithm will learn faster.

To initialize the first hypothesis h_0 domain theory B is used so the hypothesis is consistent with the theory B. With this initial hypothesis the algorithm starts to learn the training examples. Therefore, inductive learning is used. When there are inconsistencies between the data and the theory, the hypothesis is refined. When there are no inconsistencies, the output hypothesis will be the same as the initial hypothesis. When using analytical learning to initialize h_0 the algorithm is

more likely to find a final hypothesis that fits the theory and has a better generalization accuracy, but only when the theory is approximately complete and correct.

KBANN (knowledge-based artificial neural network) is an example of this method. It uses Horn clauses as prior knowledge to initialize the interconnections and weights in a neural network. This type of networks will perfectly match the domain theory *B*. In order to refine the network, backpropagation (see glossary) is used to learn the data. When the inconsistencies between the theory and data are small, the algorithm is able to learn a hypothesis that fits the domain theory with errors smaller than when only backpropagation is used. However when they are large, the algorithm is not able to learn the concept. In that case it's better not to initialize the network and just use backpropagation (T.M. Mitchell, 1997).

For the problem of tuning the parameter set, prior knowledge can't be used this way, because the problem to be learned isn't based on some first-order if-then rules or neural networks based on them. The problem is to tune the parameter set (see previous section). Also there is no theory to set those parameters at once, otherwise no learning algorithm would be needed. But when tuning by hand, you also start with an initial parameter set that is based on the acquired knowledge. This can still be done using a learning algorithm. Due to the initialized starting hypothesis instead of a randomly chosen one, the application will match the cases better already in the first learning cycle, so the algorithm can converge faster to an optimal parameter set than when using a randomly generated starting hypothesis. This parameter set probably fits the requirements of the client in a better way because they are taken into account using the initial hypothesis. The properties that were emphasized with high weights, will still have higher weights, but will be refined while learning. This also counts for small and negative weights.

The use of an initial hypothesis is also useful for the algorithms decision tree learning and genetic algorithms: it will also help them to learn more accurately. Using analytical learning also provides an advance considering the data, because we need less trainings and test cases than with only inductive learning.

4.1.2. Knowledge applied to the Tuning Problem

Using prior knowledge to learn a concept is very effective, provided that the used knowledge is correct and complete. However, the knowledge of the tuning experts (prior knowledge) can't be used as a theory for another reason (see previous (sub)sections), so it must be used in another way. To use it, the implicit knowledge must be made explicit (see below). After that, it can be used as a knowledge-based function, which will be a part of the learning algorithm. With this knowledge-based function *f* and information provided by the cases, weights, types and errors, the parameters can be learned. It is chosen to use a function based on tuning knowledge so the knowledge is a part of the algorithm and the algorithm will become domain-independent. This domain-independence can only be achieved if such tuning knowledge exists and the function is domain-independent.

Figure 4.1 on the next page shows the diagram of the knowledge-based learning algorithm and table 4.1 shows the short pseudo code. Here the algorithm is explained. First the parameters are initialized using prior knowledge. Then the cases the client has made are input to Elise. Elise also uses the DataDictionary, which contains the actual parameters. Elise calculates the match scores and gives them as output. The output of each case is compared to the target score, which is determined by the client together with the training cases. There can be a difference in the match percentage (positive or negative) but also in the fact that a MATCH is evaluated as a NOMATCH or the other way around (scores above a certain threshold result in a MATCH and scores below in a NOMATCH, but this can only be evaluated when the threshold can be derived from the training examples, otherwise the threshold will be set after learning). This difference results in the error ε_i,

which is input for the knowledge-based function *f*. This function also gets the actual parameter settings, and information from the training cases as input. Based on the knowledge-based rules (a few examples are given below) the function searches for the "most striking error" in the parameter set, so it can be updated (see 2.3 on Tuning). When an update has taken place, a new cycle starts. This continues until the algorithm has found an optimal parameter set and terminates. The algorithm has found an optimal parameter set when the matches are calculated as matches and the nomatches as nomatches (when the threshold is known) and when all the errors lie between –x% and +x%.

The information provided by the training examples contains match scores of the different properties. For every case all the demanded properties of the vacancy deal are compared to the offered properties of the resume deal and the other way around. This results in the same detail match scores as Elise would calculate. The scores are linked to the case it belongs to, so the knowledge-based function knows which scores are related to a certain error. Those scores have to be calculated only once.

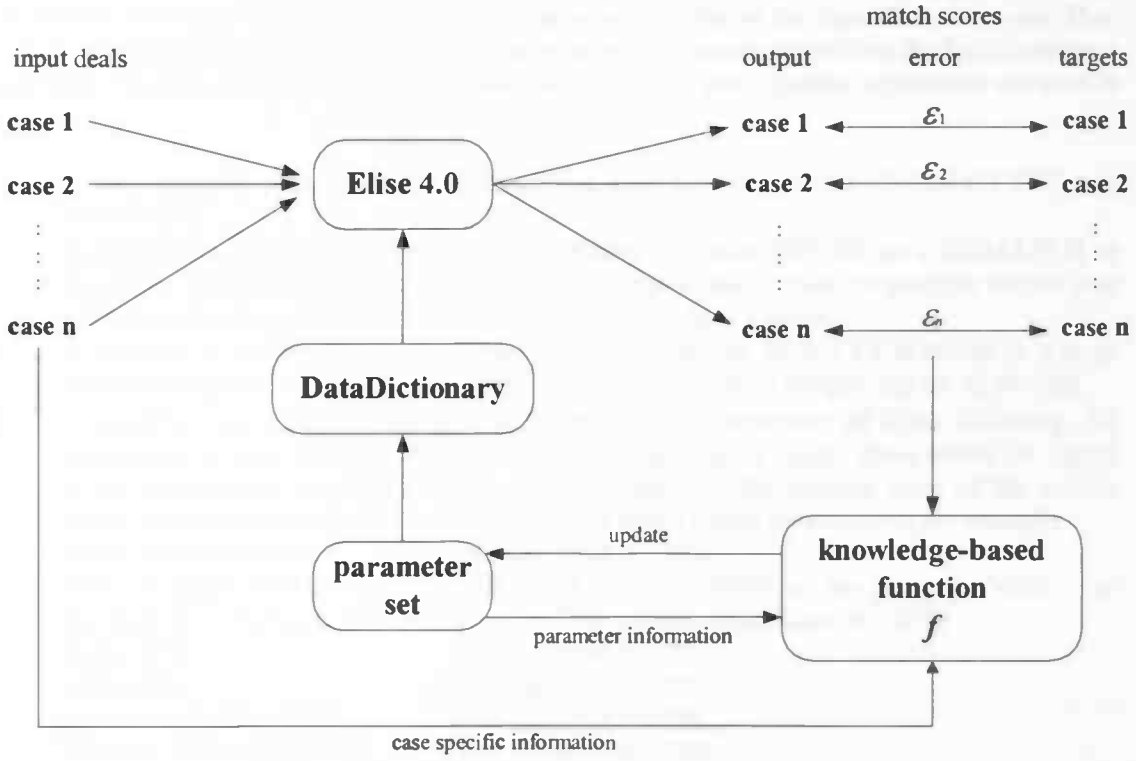


Figure 4.1. *Diagram of Knowledge-based Learning Algorithm*

Knowledge-based Learning

1. Initialize the parameters (see section 2.3 Tuning)
2. Run Elise with the actual parameter set and input cases
3. Calculate the error between output en target match score
4. Use the knowledge-based function *f* to update the parameters
5. Repeat steps 2, 3 and 4 as many times till the termination criterion is met

termination criterion: all cases must match right, so matches must score above the threshold and nomatches below, besides that the scores must be within a range (about 5% above or below) of the target-score

Table 4.1. *Knowledge-based Learning algorithm*

The knowledge-based function *f* is the essential part of the algorithm. The most important task of this function *f* is to update the parameters in such a way that the algorithm converges towards an acceptable parameter setting. Below some of the possible rules of the algorithm are given. They contain specific tuning knowledge, but also knowledge of characteristics from the DataDictionary and Elise. The first two rules can also be used as constraints when genetic algorithms are used to tune.

- Often, NEVER and ALWAYS shouldn't be used together in one PRODUCT.DIC (see section 2.3).
- Sometimes it is clear that a property should have the value NEVER on a NOMATCH or the value AND on a MultiInstance (see 3.1). In such a case, it must be possible for the user to fix the specific parameter, so the algorithm isn't able to change it.
- A NEVER is closer to a negative weight or zero and an ALWAYS is closer to a large positive weight, so the algorithm shouldn't change a negative weight into an ALWAYS.
- A possible way to find the "most striking error" in the parameter set is the following. All cases have an error between the target and the output match score. These errors are linked to the used match properties of that case. Per property the average error of the related match scores can be calculated for the different detail match percentages, for example:
  skills 100% (MATCH) average error is –20%
  (For all cases with an actual detail match score of 100% on the property 'skills', the average error between the target score and the actual match score is –20%)
  skills 60% average error is –10%
  skills 25% average error is –5%
  skills 0% (NOMATCH) average error is +5%
  function 100% (MATCH) average error is –35%
  function 40% average error is –20%
  function 0% (NOMATCH) average error is –5%
  Looking at the properties and the average error it is clear that the property 'function' has the biggest error (and the error is bigger when the match is better). It's negative, so the MATCH WEIght should be increased by the algorithm in proportion to the error percentage (the precise delta function for the weight update should be determined by an investigation).
- To gather rules to find the "most striking error" and relate updates to errors, a tuning expert should be interviewed in depth. Those rules will be a part of the knowledge-based function.

4.1.3. Summary

Analytical learning can have various advantages on inductive learning, but only when the prior

knowledge is approximately correct and complete. The problem of tuning the parameters is that standard analytical learning can't be used, so an alternative is described above. This alternative is introduced as knowledge-based learning and uses the knowledge-based function f to update the parameters. The algorithm also uses analytical learning to initialize hypothesis h_0, so fewer training data is needed. It can also be generalized to other domains. The function can handle different numbers of properties and cases, so it is of no importance whether the domain consists of twenty properties and thirty cases or fifty properties and one hundred cases. The meaning of the properties is also of no importance, so the domain can be resume vacancy matching or criminal matching. So generalizability and the number of needed training examples are two important advantages. Due to those advantages the time factor is also reduced, compared to tuning by hand.

A disadvantage is the possibility to extend the learning possibilities of the algorithm. When new features are implemented in Elise, the parameters related to those new features must be learned. However, no knowledge is available to tune the new parts, so it is hard to change the knowledge function. Another disadvantage can be the robustness of the algorithm: it is unknown if the algorithm converges to an optimal weight set. The rules on which the knowledge-based function f is based haven't been investigated or tested. So it is not yet known if the knowledge from tuning by hand can be captured in a rule set which can balance the parameters as well. This is an interesting case for a future investigation.

4.2. Genetic Algorithms

Genetic Algorithms are based on the natural evolution theory of Darwin and developed by John Holland in the seventies. They are blind search algorithms that evolve, based on the survival of the fittest principle, towards the target (is fittest) solution (for the tuning problem the optimal weight set). Genetic algorithms are robust (in complex spaces), stochastic and random. Those are advantages. A disadvantage is the computation time, which is longer than other learning algorithms. Genetic operators take care of the evolution. They recombine strings based on their performance: if the performance is good, the probability of selection (stochastic part) is also good, and thus the reproduction chances are higher (but still random) (D.E Goldberg, 1989). Genetic algorithms are optimization methods. It is not guaranteed they find an optimal solution, but they often succeed in finding a solution with high fitness (T.M. Mitchell, 1997).

In table 4.2 an overview is given of the comparison of natural and genetic terminology (D.E. Goldberg, 1989).

Natural Evolution	Genetic Algorithms
chromosome (total genetic prescription for the construction and operation of some organism)	string (all the bits)
gene (part of a chromosome)	feature, character, or detector (collection of bits)
allele (one value of a gene)	feature value (individual bit)
locus (position of a gene)	string position
genotype (total genetic package)	structure
phenotype (interaction of the total genetic package with the environment)	parameter set, alternative solution, a decoded structure

Table 4.2. *Comparison of Natural and Genetic Terminology*

In the following sub-sections the basic subjects of genetic algorithms are discussed. First the

population, generations and fitness are discussed, then the operators and at last different kinds of encoding. After this global description, the algorithm is applied to the tuning problem and a summary is given.

4.2.1. Population, Generations and Fitness

Like evolution, survival of the fittest also is the main principle of a genetic algorithm. There is a population that exists of strings and there is a target to learn. Each string is one hypothesis of the hypothesis space. This string can perform well (it's close to the target), or not so well (it's not close to the target). This is called the fitness of a string and is calculated by a fitness function. Based on this fitness and a random factor, strings can be reproduced or not (see 4.3.2. Operators). Each time when reproduction has taken place, a new offspring is created and a new generation is born. The first generation is the initial hypothesis population and the last one is the one that has evolved towards the target, which means: contains a string that approximates the target (fitness is 100% minus a certain error). An example of a fitness function $f(x) = x/2$ and calculated fitness scores is shown in table 4.3 below where the value of x is represented by the string.

String No.	Binary String	x	fitness $f(x) = x / 2$	percentage of total fitness
1	1 0 1 1 0 0 0	88	44	44 %
2	0 1 0 1 0 1 0	42	21	21 %
3	0 0 1 1 1 0 0	28	14	14 %
4	0 0 1 1 1 0 0	18	9	9 %
5	0 0 0 1 1 1 0	14	7	7 %
6	0 0 0 1 0 1 0	10	5	5 %
Sum			100	100 %

Table 4.3. *Fitness Function with calculated Fitness Scores for six Strings*

The next section will handle the basic operators that evolve the strings to the next generation based on their actual fitness score.

4.2.2. Operators

Operators used by genetic algorithms take care of the evolution. It is already mentioned that they work randomly, but the discovery of a solution is not all pure chance. This is implied by mathematician J. Hadamard in 1949. By building the new population based on the best parts of the last generation, they guide the algorithm towards better solutions. It is a combination of a structured, and randomized process, which is the power of the algorithm.

There are various operators, but genetic algorithms namely use three basic operators. Those are: reproduction, crossover and mutation. There are also other lower and higher level operators. Below a short description of the basic operators is given.

4.2.2.1. Reproduction

Reproduction is an operator that takes care of the process of selecting strings that will be copied for the following population. This selection is based on the fitness function. Each string in the population has certain fitness. When a string has a high fitness, its chances of reproduction are higher than when its fitness is low. A roulette wheel is a simple way to implement the selection for reproduction. Each string gets a part of the wheel. The size of that part is related to its fitness (of each string the fitness percentage is calculated). Then the wheel is spun n times (n is the number of strings in the next generation). Every time the wheel stops at a certain string, that

string will be reproduced. Figure 4.2 shows a roulette wheel with six different strings. The percentages were calculated in table 4.4.

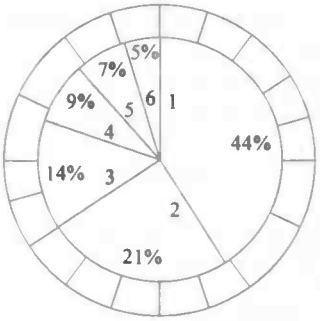


Figure 4.2. *Roulette Wheel*

4.2.2.2. Crossover

When the selection procedure has taken place, random pairs of strings mate and new offspring is born. While mating a random crossover point (the sign |) is chosen (or a number of points), and the substrings after the pipe are swapped between the strings. An example is the following:

```
A = 0 1 0 1 1 1 0 | 0 1 0 1 1
B = 1 0 0 0 1 1 0 | 1 1 0 0 1

becomes

A' = 0 1 0 1 1 1 0 1 1 0 0 1
B' = 1 0 0 0 1 1 0 0 1 0 1 1
```

4.2.2.3. Mutation

When only reproduction and crossover are used, the genetic material loses information. 1's or 0's at certain locations in a string can disappear. To prevent this loss, mutation changes bits in a string randomly. A 1 becomes a 0, and a 0 becomes a 1. It is only used sparingly, just like in natural evolution.

4.2.3. Encoding

Each string must contain information about the solution it represents. This information can be encoded in several ways. Below four types of encoding are described.

4.2.3.1. Binary Encoding

The encoding type binary encoding is mostly used. Every string is composed out of 0's and 1's (see below). For some problems this type is not practical, then one of the next encoding types can be used.

```
A = 0 1 0 1 1 1 0 0 1 0 1 1
B = 1 0 0 0 1 1 0 1 1 0 0 1
```

The third value besides a 0 or a 1 is a # or a *, which stands for a "don't care." When an allele has that value, it matches a 1 and a 0. So the string 0 * 1 matches 0 0 1 and 0 1 1.

4.2.3.2. Permutation Encoding

Using permutation encoding the order of alleles of the string is important, so it's used for ordering problems like the traveling salesman problem. The sequence of the numbers is the

solution to the problem. There is a risk that a number disappears from the string and another one is duplicated. This means that the string loses information and becomes inconsistent with the problem. To avoid this some changes to the crossover and mutation operator have to be made. An example of crossover is:

```
A = 2 1 5 6 | 8 4 9 3 7
B = 3 8 6 5 | 4 9 7 2 1

becomes

A' = 2 1 5 6 3 8 4 9 7
B' = 3 8 6 5 2 1 4 9 7
```

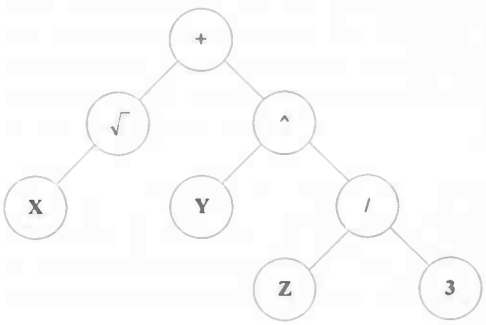
4.2.3.3. Value Encoding

When using value encoding each string contains some values. These values can be integers, reals, chars, or more complex objects like *yellow*. When learning such complicated values it is desirable to use this type of encoding, because binary encoding is difficult for such problems. When using this encoding type it is also necessary to make changes to the mutation operator (for example add or subtract small values of a real or integer) because the standard operator function won't work well due to the different representation. Below an example of value encoding is given.

```
A = F G A E B D D F G G F E or
B = 3.8947 5.8971 2.0936
```

4.2.3.4. Tree Encoding

Not only strings are used as representations, but also trees. Trees are very useful when learning mathematical functions or evolving programs (often Lisp programs). Operators now don't use a position in a string to crossover or mutate, but a node in the three. In figure 4.3 an example of tree encoding is given.



produces the function : $\sqrt{x} + y^{z/3}$

Figure 4.3. *Tree Encoding*

4.2.4. Genetics applied to the Tuning Problem

The subjects discussed above (4.2.1, 4.2.2 and 4.2.3) are applied to the tuning algorithm in this subsection. Analytical learning as described in section 4.1 will also be discussed in this section.

Each string of the population will contain information of the parameters to be tuned. For the representation of the string a combination of binary and value encoding is appropriate, which is explained below.

Of each property the WEIghts can be a NEVER, an ALWAYS or an integer, a MultiInstance can

be EXClusive, REUse or OR and a MultiValue can be OR, AND or INTersection. For each parameter only one of the three options can be active, which means that only a NEVER *or* an ALWAYS *or* an integer can be set for a WEIght parameter, a MultiInstance can only be set to EXClusive *or* REUse *or* OR and a MultiValue can only be set to OR *or* AND *or* Intersection. This can be represented in a string of three bits (binary encoding) for each parameter of each property. The bits will represent the three possible values and the value that is active will be represented as a **1** and the two values that are inactive will be represented as two **0**'s. It is also possible to fix a parameter when it doesn't have to be learned. This can be done using #-signs instead of bits, which means that the algorithm isn't allowed to change the parameter.

When an integer of a noobject, novalue, nomatch or a match Weight is active, there must be a numeric value that is related to the specific integer. This can be done using value encoding for a separate part of the representation string. To relate each numeric value to the corresponding integer bit, a link is made for each pair. An example of the representation string is given in figure 4.4.

Figure 4.4. *Substring Encoding*

So each bit and each numeric value contains information about the parameter set that must be learned, because it is related to a specific value of a specific property. This means that the structure of the string is very strict. The length of the string and the substring is known because it is related to the number of properties. Thus, information can be extracted easily to update the DataDictionary.

Although the structure is strict, the algorithm can easily be applied to different domains, because only the number of bits and numeric values will vary. The meaning is of no importance, because the algorithm can decode the string to the related properties. If Elise is extended with extra features that must be learned, the algorithm can easily be changed, because only extra decoding functionality is needed. However, this is only the case when the new features can be represented as bits or numeric values.

When new offspring is created, the operators must take the structure of the string into account, otherwise the structure of the string becomes inconsistent with the problem and no meaning can be derived from it anymore. Thus, when using this representation special crossover and mutation functions are necessary, so that substrings and links will be maintained.

Besides the representation and attached meaning, prior knowledge can play a role. It was already mentioned in the section on knowledge-based learning that initializing the hypothesis could also be used for a genetic algorithm. In this case one hypothesis is not enough, because the algorithm starts with a whole population. This population contains a number of hypotheses (strings), so not only one initialization must be made, but several variations (until the size of the population is filled, which will be determined in a later stage).

Also other knowledge of the domain can be used. Sometimes it is known that a weight must be a NEVER (see 2.3 on tuning). Then the substring containing that information could be disabled so the algorithm knows it may not change that substring.

The algorithm will terminate when an optimal weight set is reached. This is when most of the actual matches are calculated as matches and most of the actual nomatches as nomatches. Also

most of the total scores must lie within a range from the target scores (target score $case_i$ – x% < actual score $case_i$ < score + x %). In table 4.4 the steps of the algorithm are shortly described.

The role of Elise in the algorithm is calculating the match scores of the cases and then reporting the results to the algorithm, so the fitness score of the strings can be calculated. This will happen each time Elise gets string information from the algorithm for the parameters in the DataDictionary.

Genetic Algorithm

1. Initialize the first population
2. For each string in the population calculate the match scores of all cases with Elise 4.0
3. Calculate the fitness of each string in the population based on the difference between the match scores and the target scores of the cases
4. Create new offspring for the next generation with the operators
5. Repeat steps 2, 3 and 4 as many times till the termination criterion is met

termination criterion: most cases must match right, so matches must score above the threshold and nomatches below, besides that most scores must lie within a range (about 5% above or below) of the target-score

Table 4.4. *Genetic Algorithm*

4.2.5. Summary

Genetic algorithms are expected to be able to tune the parameters. Due to the various possible encoding types, the different kind of values (like NEVER, an integer or a fixed parameter) can be learned. Because of the robustness of the algorithm, the algorithm probably will converge to a parameter set with high fitness. Also the representation is relatively easy, due to the different types of encoding that can be combined. Using analytical learning to initialize the population, the algorithm will converge faster towards an optimal weight set with fewer training cases than without analytical learning. The ability of generalization to other domains and the possibility to adapt the algorithm to extended features in Elise is also a pro of this algorithm.

Small difficulties are the changes that have to be made to the commonly used crossover and mutation operators, because of the structure of the string. And a disadvantage of genetic algorithms is their computational time (see beginning of this section). But with the speed of today's computers, this is not really a problem. So genetic algorithms are expected to be successful tuning the DataDictionary of Elise.

4.3. Decision Tree Learning

First a general introduction on decision tree learning is given, which is based on the article "Induction of Decision Trees" (J.R. Quinlan, 1986) and the chapter on "Decision Tree Learning" of the book "Machine Learning" (T.M. Mitchell, 1997). After that, decision tree learning is applied to the tuning problem and a short summary is given.

Decision tree learning is a robust algorithm that must learn a classification task. To learn this task, the algorithm will create a decision tree that is able to classify each object that is presented to the algorithm. To create the tree, a universe of objects (the training set) will be presented to the algorithm. Each object will belong to one class, so the classes must be mutually exclusive. The

algorithm must learn the procedural knowledge necessary to classify the objects and this knowledge will be represented as a decision tree.

The training set presented to the algorithm is a relevant part of the learning process because the tree is built based on the frequency of information in the examples. Each object in the set consists of a number of attributes and each attribute represents a feature that has one value of the set of discrete, mutually exclusive values. In table 4.5 an example of a training set is given. The classification task is related to the weather and to play tennis or not and the set contains twelve objects of Saturday mornings. Each object contains four attributes (outlook, temperature, humidity and windy) and a classification (*P* or *N*, for play tennis or not). The example is a two-class induction task (*P* and *N*: positive and negative instances), but multiple classes are also possible. The related decision tree is shown in figure 4.5. The leaves represent the classes, and the other nodes represent the attributes.

No.	Attributes				Class
	Outlook	Temperature	Humidity	Windy	
1	sunny	hot	high	false	N
2	sunny	hot	high	true	N
3	overcast	hot	high	false	P
4	rain	mild	high	false	P
5	rain	cool	normal	false	P
6	rain	cool	normal	true	N
7	overcast	cool	normal	true	P
8	sunny	mild	high	false	N
9	sunny	cool	normal	false	P
10	rain	mild	normal	false	P
11	sunny	mild	normal	true	P
12	overcast	mild	high	true	P
13	overcast	hot	normal	false	P
14	rain	mild	high	true	N

Table 4.5. *Training Set (Quinlan, 1986)*

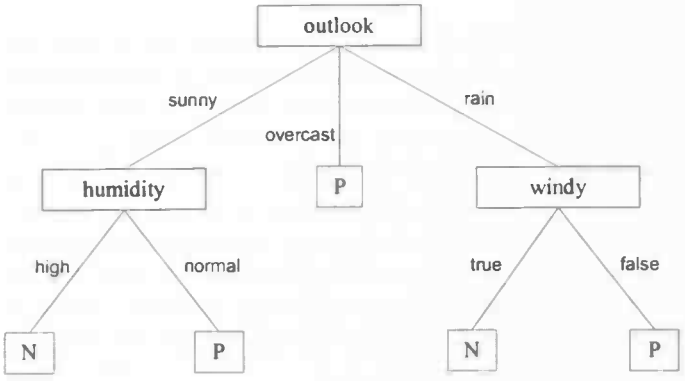


Figure 4.5. *Decision Tree (Quinlan, 1986)*

An assumption to learn a correct decision tree is that the attributes of the objects are adequate,

which means that the cases must be consistent and correctly classified, so equal objects are always classified to the same class. If this is the case, the algorithm can learn a correct decision tree and most of the time several correct decision trees. If more trees can be constructed, simple (small) trees are preferred above more complex (large) trees because these are more likely to represent the underlying structure (Occam's Razor, see glossary). Complex trees are often a reflection of the training set, while simple trees are suspected to classify all objects correctly.

If the training set is not adequate (the attributes contain noise or are incomplete), decision tree learning is still able to solve the problem up to a certain level (Quinlan, 1986). However, this will not be elaborated in this paper.

There are several decision tree learning algorithms that build the tree in different ways. One of them is ID3. ID3 constructs the tree top-down. It starts to "ask" which attribute should be the root. Therefore it calculates the information gain of each attribute and the one with the highest gain is selected. This attribute classifies the object best for the node. For each node this question and calculation are repeated and when the tree can classify all objects, the algorithm terminates.

4.3.1. Decision Trees applied to the Tuning Problem

Decision tree learning as explained in the previous section is not appropriate to learn the tuning problem. In the first place, it is not a classification task. Secondly, Elise calculates actual match scores based on the deals of the training cases, so there are no objects like "Saturday mornings" that must be analyzed by the algorithm. Therefore, changes must be made to the algorithm to apply decision trees to the tuning problem. Below a proposal will be given. It will not be described in detail, because too many changes must be made before the algorithm will be appropriate to learn the problem. So, first a description of the representation of the objects is given and a proposal to use those objects is made. After that, a possible decision tree is described.

To determine the representation of the objects, it is important to know what information is available. First of all, the training cases with the target matches are given, secondly, the actual parameter set is known and at last the properties that must be learned are in the PRODUCT.DIC files. With this information Elise can calculate the actual match scores, which can be compared to the targets to measure the performance and see if the algorithm has solved the problem. This comparison gives the most valuable information because it is related to the performance of Elise with the actual parameter set. So the objects will be the difference between the target and actual match scores. The objects can be on match score level and the attributes on property (match) level with the actual values as values. The exact representation will be left for further research.

The task of the decision tree is tuning the DataDictionary of Elise. To tune the DataDictionary, changes must be made to the parameters in the PRODUCT.DIC files. To combine the tuning process with a decision tree, the structure of the tree is important. When tuning by hand, targets are compared to actual results. If they are equal, the tuning process can be finished. If not, the most obvious error is searched to determine which parameter must be changed (see section 2.3). This structure can be maintained using the decision tree. So the root (first node) must check whether the targets are approximated (the differences between the target and actual matches are smaller than x). When this is the case, the algorithm will terminate. Otherwise, the tree will be searched based on the target cases compared with the actual match result. When the tree is searched, the algorithm will find a leave at the end of the tree that must make a proposal to update a parameter.

There must be two different kinds of leaves, namely discrete leaves and dynamic leaves. The discrete leaves will be qualified to change the four WEIght and the two TYPe parameters into respectively NEVER or ALWAYS and EXClusive, REUse or AND and AND, OR or

INTersection. The dynamic leaves are only qualified to update a numeric value of a WEIght parameter. For the update, a function is used to determine the value that must be added to or subtracted from the numeric value. Therefore the function must use the differences between the target and actual match scores. Further, when a parameter is fixed, the tree must have no leave suggesting an update to that parameter.

Between the first node and the leaves, the tree must guide the algorithm to the "most striking error" and the related necessary update. This structure of the tree must be created before learning starts. So the algorithm will follow two steps. First, it will create the tree and secondly, it will learn the tuning problem. Creating the tree can be done by hand, but this will provide no time advantage (which is a requirement). It can also be learned based on the domain dependent properties, the target cases and the actual match result, but this issue will be left for future research.

4.3.2. Summary

To develop a decision tree that can learn the tuning problem, a lot of research must be done. For this research, no time is available in this project, so only a proposal is made. Based on this proposal the chance on success cannot be determined, but assumptions can be made. All the types of parameters can be learned and because the algorithm will create the tree by itself based on the domain, the algorithm will be generalizable to different domains.

Besides the assumptions, there are also questions. Like the amount of needed training and test data, the possibility to learn other parameters and will there be a time profit? To let the algorithm learn new parameters when Elise is extended, will probably be a lot of work. Cause different parameters must be learned, which will probably behave different than the WEIght and TYPe parameters, different decisions for those parameters must be made.

4.4. Conclusion

All the algorithms have their pros and cons, but analytical learning is interesting for all three of them. The algorithms can be combined with prior knowledge of the matching domain to initialize the starting hypothesis h_0. This means the algorithm can learn more accurately, and needs less training cases than it would otherwise (see subsection 4.1.1). The number of training cases probably should be increased compared to the number of cases used by manually tuning, but this will be investigated in the next chapter. This brings us to the conclusion that analytical learning will be combined with the selected algorithm.

To compare all the pros and cons of the three algorithms, table 4.6 is given on the next page. It represents a weighted comparison related to the general requirements mentioned in chapter 3. Of each algorithm the requirements are discussed in the summary of the related section. The requirements are translated into a score (--, -, +/-, + and ++). Each requirement is also weighted by the importance for Bolesian and AI/TCW.

This comparison shows that a genetic algorithm is the most appropriate learning algorithm to solve the tuning problem. It is the only algorithm that scores plus signs on all the requirements. It is the most robust algorithm, the algorithm can be generalized for different clients and domains, and it is the most flexible related to changes or expansions in the tuning domain. All these arguments give the algorithm the highest chances to learn an acceptable parameter set. So a genetic algorithm is the learning algorithm, which will be implemented and tested for the tuning problem.

Properties:	Importance: (at the top Bolesian, under AI/TCW)	Genetic Algorithms	Knowledge-based Learning	Decision Tree Learning
1. Generalizability	++ / +	++	+	+
2. Scientifical Value	+/- / ++	+	++	++
3. Change on Success	++ / +	++	+/-	+/-
4. Amount of Needed Training and Test Data	++ / +	+	++	+/-
5. Robustness	++ / +	++	+/-	+
6. Time Profit	++ / +/-	+	+	+/-
7. Possibility of Learning Extra Parameters	++ / +	+	--	-
Total:		++	+	+/-

The total score is calculated as follows: ++ = 2, + = 1, +/- = 0, - = -1, -- = -2, the valuation of Bolesian is two times higher than the valuation of AI/TCW, the maximum total value is 66 (++), and the minimum value is –66 (--).

Table 4.6. *Weight of Pros and Cons of the three Selected Learning Algorithms*

5. Design of Genetic Algorithm

By now the learning algorithm is selected, the requirements, basic assumptions, and limitations are set, and the expectations are clear. So it's time to specify the complete design of the learning algorithm in greater detail. This design will serve as a guideline for the implementation of the genetic algorithm. In this chapter it will be discussed. First the global learning scheme will be presented, and then it will be worked out in detail. In section 5.2 the structure of the training and test data is outlined. In section 5.3 the representation is described based on the encoding types defined earlier in section 4.3.3. In 5.4 the size of the search space is calculated. Section 5.5 gives a definition of the fitness function, and the termination criteria are discussed in section 5.6. Section 5.7 discusses the operators, which will be used by the algorithm. In 5.8 issues concerning the population are discussed, in 5.9 the values of the control parameters are given, and in 5.10 the technical aspects are outlined. All those components together form the complete design that will be the foundation of the implementation.

5.1. Global Learning Scheme

The genetic algorithm will follow the global learning scheme while learning. This scheme is presented as a flow chart in figure 5.1. First the algorithm initializes the initial population with *n* strings (see section 5.9), each string containing all the parameters that must be tuned. From this point evolution begins. First one string at a time is selected. Second the DataDictionary is updated based on those parameters and the database is reloaded (step three). With the new parameter setting and the training cases, Elise is executed in order to calculate the match scores (step four). The training cases are presented to Elise in advance, and are stored in the database of Elise. After the actual match scores have been calculated, they are used together with the target scores (which are derived from the training cases in advance) to calculate the fitness score of a specific string (step five), using the fitness function (see section 5.5). These five steps are repeated *n* times, until the fitness scores of all strings are calculated. When all the fitness scores are calculated, the algorithm checks if the termination criterion is met (section 5.6). If it is not met the algorithm evolves within the search space (see section 5.4). It uses the reproduction, crossover, and mutation functions to create new offspring (section 5.7) until the new population is as large as the old population. Then the new one replaces the old population, and the whole process is repeated, until the termination criterion is met. When this is the case, the algorithm will test the solution with the test cases. If the solution is not sufficient (the termination criterion is not met using the test cases), the algorithm will give the advice to check if the requirements and basic assumptions are met. If the solution is sufficient (the termination criterion is also met using the test cases) the algorithm will return the fittest string as output, which is the optimal parameter set.

It is also possible that the strings don't improve any further at a certain point, so the termination criterion will never be met. If this is the case, or the maximum number of trainings cycles is met (set before learning starts), the algorithm must also terminate with the advice to check if the requirements and basic assumptions are met.

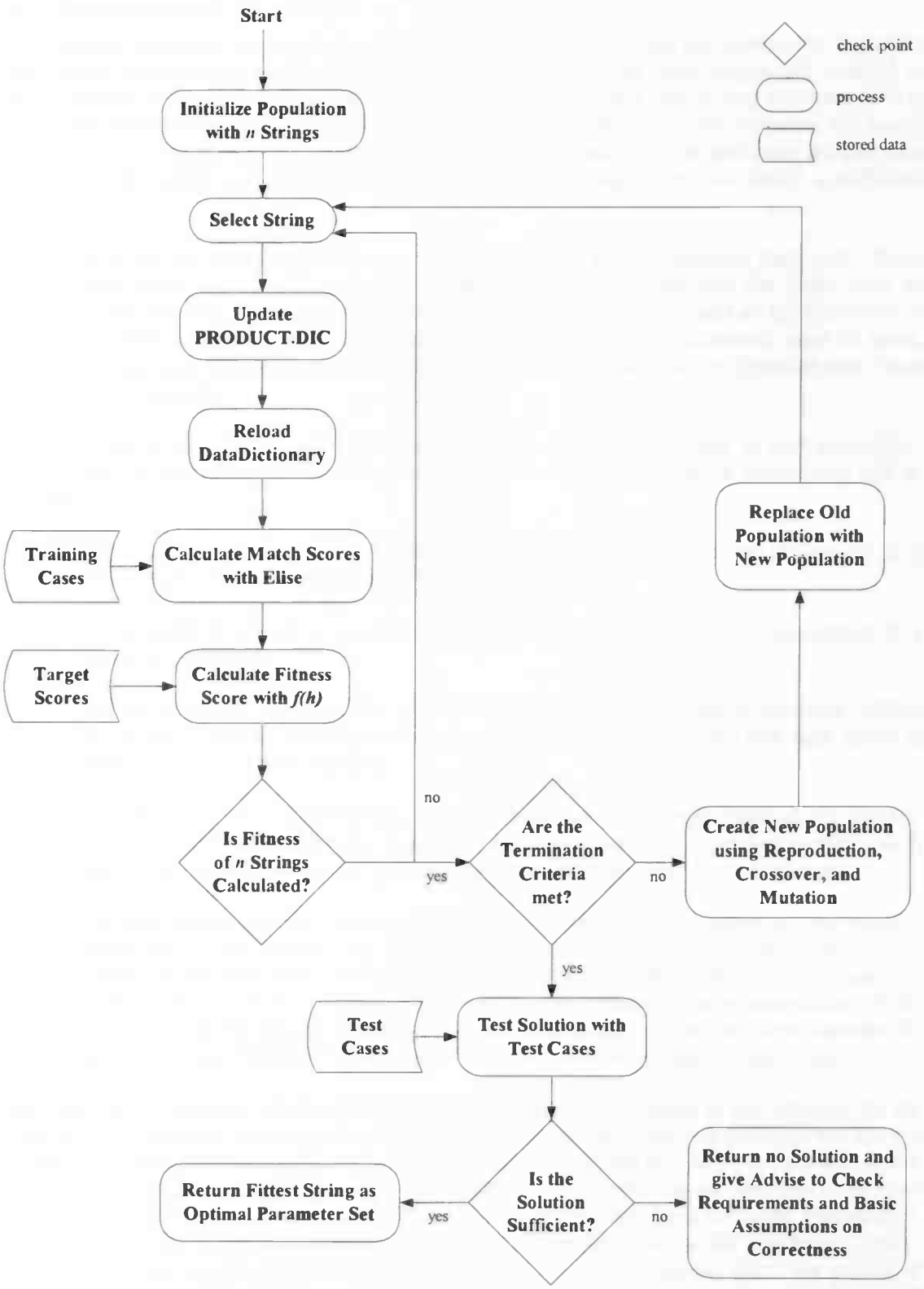


Figure 5.1. *Global Learning Scheme*

5.2. Training and Test Data

The learning algorithm needs training cases to learn from. Those cases are input to the algorithm and contain target match results that must be approached. When those targets are reached or nearly reached (see termination criterion), the algorithm terminates and the test cases are used to check the performance of the learned parameter set. Therefore Elise must calculate the match scores of the test cases with that specific solution. So if the matches of the test cases are also near enough to the target (see termination criterion again) the algorithm has found a sufficient solution.

The training and test data must fulfill some requirements to let the algorithm learn well. Those requirements were mentioned in section 2.3 and the main issues are that the cases must be consistent and real life, they must be provided with target match results, and all the properties of the domain must be covered by at least one case. One complement requirement must be made, namely that the cases must contain both matches and nomatches because the algorithm must learn to match both correctly.

The provision of the target match results can be done in five different ways. In each alternative, the judgment of the cases differs. All are discussed below and one of those alternatives will be selected.

1. Every match within a case is linked to an exact total match score (expressed as a percentage) and is designated as a *match* or a *nomatch*.

2. Every match is related to a range (like 70%-80% or 80%-90%) and is designated as a *match* or a *nomatch*.

3. Within each case only the order of preference between the matches is specified. Within this order it must be possible to value two matches equally. Besides that each match is marked as a *match* or a *nomatch*.

4. A combination of the previous two types: there is a predefined order between the matches within a case (within this order it must be possible to value two matches equally), each match must belong to a range, and each case is marked as a *match* or a *nomatch*.

5. The total match score is a weighted average of the resume match score and the vacancy match score (see section 2.1). The algorithm can learn from those two match scores instead of the total score. For these, one of the four alternatives above can be used and both matches must be designated as a match or a nomatch. The representation of the cases for both the resume and the vacancy match scores will be the same because the structure of the PRODUCT.DIC files is equal and must be learned the same way.

The important differences between the above five alternatives are related to the difficulty for the client to judge the cases, the need of learning the threshold and the expected quality of the learned solution. The quality of the solution is related to the usability of that solution. For each alternative the algorithm can learn a solution close to the target but not all solutions are desirable or in other words usable because not all the score information can be extracted from the cases. For example when only the order of the matches is known there is no information about the total match scores. Most clients want to approach exact scores so when the targets are defined exact, the quality of the solution will be high. So the question is which quality is satisfying enough to the client, and whether the client is able to fulfill the requirements related to the requested cases. Below the differences are discussed and the pros and cons are given for each type of representation

assuming the cases are (approximately) correct and contain no inconsistencies.

1. When all the target match percentages are known and the algorithm approximates those matches, the quality of the found solution will be high. However it is relatively difficult for the client to determine the exact percentages of the matches because usually it is hard to translate "a good match" or "a questionable match" into a percentage in a consistent way. When this type of representation is used the algorithm doesn't have to learn the threshold because it can be derived from the target cases using the ratio between the matches and nomatches.

2. The quality of the found solution will be lower than the first alternative because the matches belong to a range instead of a percentage, which is less exact. A further issue that must be taken into account is when an actual match is 89% and the target range is 90% -100%. This is considered as an error but when the match should be close to 90% it should be interpreted as correct. For the client it's about as difficult as alternative 1 to make the cases because it is also relatively hard to relate matches ("a good match") to the different ranges (70% - 80% or 80% - 90%). Using this kind of representation the threshold can also be derived from the cases using the ratio between the matches and nomatches.

3. Using alternative 3 the client will have less trouble making the cases because it is easier to determine which matches must score better than other matches within a case than to assign an exact score to each match. But the information that can be extracted is less reliable because only the order between the matches is known, not the value of a match (is the best match 85% or 95%) or the distance between different matches (5% or 20%). This means the usability of the solution will be restricted compared to alternative 1 and 2. Further using this type of representation the threshold can be learned by the algorithm or set before learning. In the first case the threshold can become very low (15%) or very high (80%). In the second case matches can be forced to score above the threshold and nomatches below, but in both cases the value of the scores are meaningless because only the sequence is learned.

4. When the sequence and the target range are known the problems mentioned in the previous two alternatives will be less, so the quality of the found solution will be higher (it will approach the quality of the first alternative). The difficulty of making those cases will increase compared to the previous two alternatives. The client must consider the order and the range. When the cases are defined, the consistency of the cases must be checked because a match higher in the sequence than another match must at least belong to the same range. For this kind of representation the threshold can also be derived from the cases using the ratio between the matches and nomatches.

5. When the alternative with two target match results is used the client must define more target results per case than when only one target match result is used. The pros and cons of the previous alternatives are the same when defining two target match results, but compared to the previous alternatives it is relatively easy to judge the cases: less properties have to be evaluated at once because only one match side must be evaluated instead of both sides. This is the reason why a relatively higher quality is to be expected. Also the usability is higher because more information can be extracted from the cases. When this alternative is used the match results will be learned separately because the match results are related to different parts of the DataDictionary. First one parameter set will be learned and then the other set so there will be two populations existing of shorter

strings. There also should be two thresholds (when necessary).

Another difference is related to the algorithm. For each different representation a different fitness function and termination criterion are needed. When the match percentages are known the errors between the target and the actual match score are the criteria that determine the fitness and termination criterion. When the sequence is known the relative position of a match compared to the position of other matches will be used as the criterion. When both are known a combination of the criteria is used. For the various criteria different information about the cases and from Elise is needed. So the differences concern the fitness function, the termination criterion, storage of the training cases, and information extraction from Elise. It is not easy to adapt the implementation from using one representation to using another so it is important to choose the representation type well.

When all arguments of all alternatives are taken into consideration option 5 combined with option 1 is the best way to represent the cases. The algorithm is expected to learn a parameter set with the highest quality because the targets will be most exact. The learned solution will also be applicable for most matching applications because of this quality. The other alternatives will be less usable, because using ranges or orders, the quality won't be sufficient. Percentages are relatively difficult to determine, but the client will be helped to define correct and consistent cases using knowledge acquisition tools. Using two target match scores instead of one it will be easier for the client to judge the cases, which is discussed before. So each case will be separated in a vacancy part and a resume part. For both parts an exact percentage will be defined for each match and each match will be designated as a match or a nomatch. The threshold will be derived from the ratio between the matches and nomatches in advance and doesn't have to be learned.

Now the representation of the cases is defined, the number of needed training and test cases must be determined. The number of cases must not be insufficient, because the requirements to cover all the properties and provide matches and nomatches must be met. When the number of cases is higher than necessary, it will not improve the learning results; it only makes it harder for the client to make the cases. A fair amount is eight till ten cases per property per training set and per test set. One case will exist of one vacancy deal linked to one resume deal with a target match result. When the vacancy PRODUCT.DIC must be learned there will be as many resume deals as properties in the vacancy PRODUCT.DIC and there will be eight till ten vacancy deals linked to the resume deal. When the resume PRODUCT.DIC must be learned, it's the other way around.

It's the job of the tuning expert to keep an eye on the consistency of the cases, the diversity of the cases (concerning the properties) and the appearance of both match and nomatch cases.

5.3. Representation

Each string of a population will represent a parameter setting. Therefore all the parameters that must be learned by the algorithm must be encoded in such a way that a string contains all the information of those parameters. The string must also be easy to decode because it must be easy to derive the parameter setting of the string.

Figure 5.2 shows the structure of the DataDictionary. There are two PRODUCT.DIC files. Both will be related to their own string population. The structure of those strings will be the same but the length can differ due to the number of properties. Each PRODUCT.DIC file contains properties, for each property parameters are defined, each parameter consists of one or more attributes and each attribute has a value. Table 5.1 shows the parameters and attributes that must be learned by the algorithm. For each property four WEIght attributes must be learned, namely the noobject, novalue, nomatch and match attribute. Each attribute can have the value NEVER,

ALWAYS or an integer between –32.000 and +32.000. Also two TYPe attributes must be learned, namely the RepeatingGroupType and the MultiValueType. The possible values for the first type are EXClusive, REUse and OR and for the second type OR, AND and INTersection. For all attributes it counts that only one of the values can be inserted to the PRODUCT.DIC file. For example only a NEVER can be the value of a nomatch weight of a specific property, not a NEVER and an integer. In other terms only one value can be active. So a string will contain the values of the attributes of the two parameters WEIght and TYPe for all properties in a specified PRODUCT.DIC file.

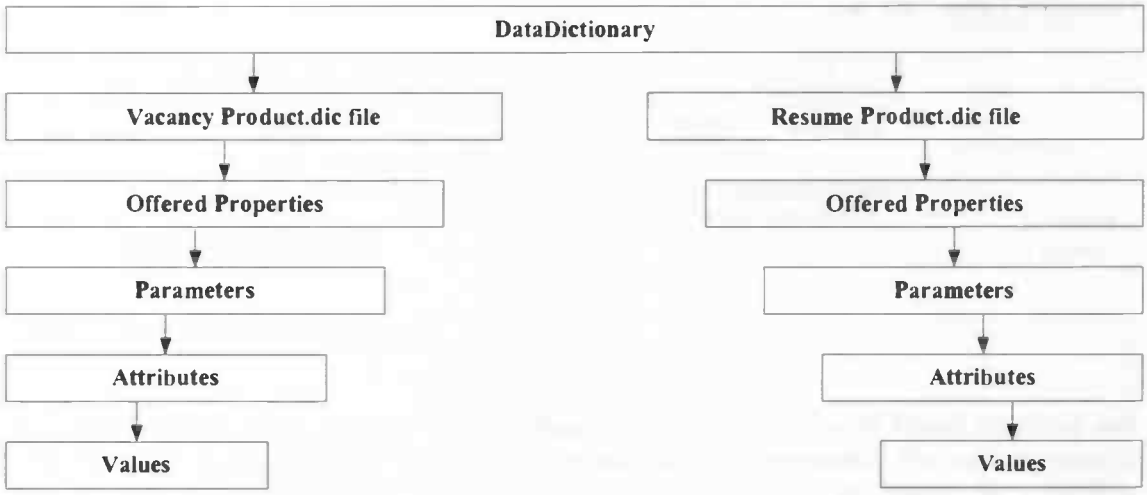


Figure 5.2. *Structure DataDictionary*

Parameters	Attributes	Possible Values		
WEIght	NoObject	NEVER	ALWAYS	-32.000 – +32.000 (integer)
	NoValue	NEVER	ALWAYS	-32.000 – +32.000 (integer)
	NoMatch	NEVER	ALWAYS	-32.000 – +32.000 (integer)
	Match	NEVER	ALWAYS	-32.000 – +32.000 (integer)
TYPe	RepeatingGroupType	EXClusive	REUse	OR
	MultiValueType	OR	AND	INTersection

Table 5.1. *Data to be Encoded per Property*

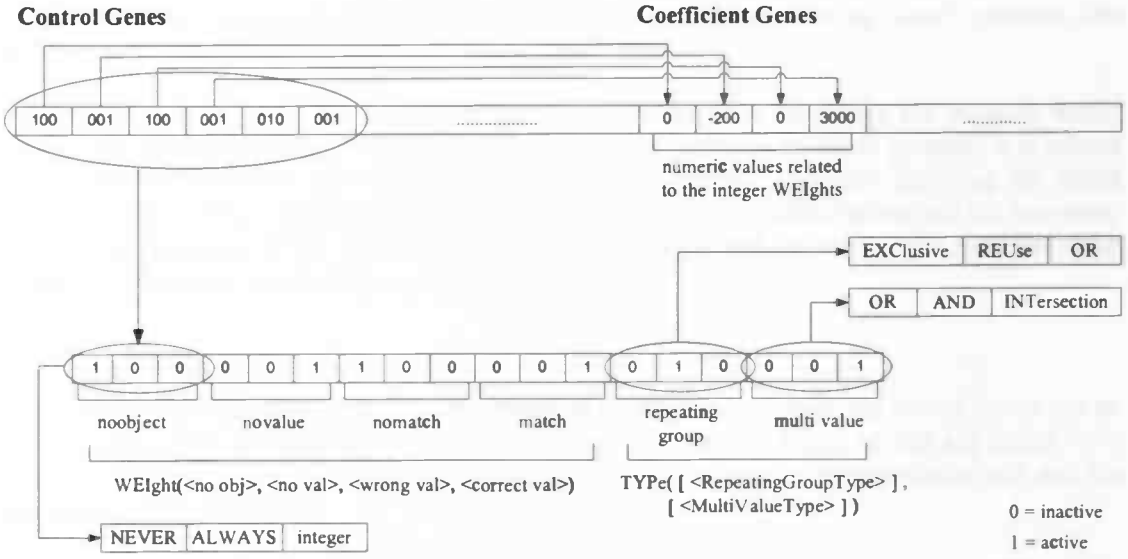


Figure 5.3. *Representation of the Strings*

The representation of the string is given in figure 5.3. A combination of binary encoding and value encoding is used because two different values must be represented. For each attribute it must be clear which value must be active (NEVER, ALWAYS or an integer for a WEIght attribute or EXClusive, REUse or OR for the RepeatingGroupType or OR, AND or INTersection for the MultiValueType). Each attribute can be represented as a substring of three bits, each bit representing one of the possible values and only one of those bits will be active (encoded as a one), and the other two will be inactive (encoded as a zero). When the integer of a WEIght attribute is active the numeric value of that integer must also be known. Therefore each WEIght substring will be related to a numeric value.

The first (binary encoded) part is composed of control genes because those genes control the activation and deactivation of the different attribute values. The length of this binary part is eighteen bits (4·3 (WEIght) + 2·3 (TYPe)) per property. The second (value encoded) part is composed of coefficient genes (their activation depends on the related control genes). Those genes are the numeric values related to the integer WEIght attributes and the length of this part of the string is four times the number of properties. The structure of this string is known as a hierarchical genetic algorithm chromosome structure (K.F. Man, K.S. Tang, S. Kwong, 1999).

Within this string each substring of three bits is called a building block and the WEIght substrings form a building block with the related numeric value. A building block represents a collection of genes that work well together. In this representation it is a collection of genes that must work together, determined by the structure of the string. The initial population contains all the building blocks that form an optimal solution, but they are spread throughout the population (and probably some numeric values must be adjusted). So the algorithm must recombine the strings and mutate the numeric values to collect (most of) the highly fit building blocks within one optimal string.

However, sometimes the algorithm must not change a parameter. For example when it appears from knowledge acquisition that a nomatch WEIght must be a NEVER or a MultiValueType must be an AND, the value should be fixed. It can also occur that a TYPe doesn't have to be learned because no multi values or multi instances are possible for the related parameter. In such

a case the related substring must be marked as fixed. This can be done using 'fixed'-symbols, like '%%%', instead of bits. Those symbols mean, "don't learn".

Another option is to delete the substrings, but then the number of substrings per property would differ. Then for each substring it must be defined to which attribute of which property it is related and numeric values must be related to the specific substrings, otherwise decoding the string would not be possible, while this is a requirement. This is a complicated option and not necessary because the substring of the fixed attributes can be disabled and the string will have the same number of substrings for all the properties.

5.4. Search Space

Based on the representation, which is introduced in the previous section, the search space can be defined. This space contains all the possible strings that can be found by the algorithm. It is important to have a clear picture of this space because it is related to the population size and the complexity of the algorithm.

When all possible strings are taken into account, the search space becomes incredibly large. Therefore, it is chosen to reduce the number of possible strings for this project, to make the testing of the genetic algorithm quicker. Reducing the number of possible values does this. At first ALWAYS is left out the learning process. This can be done because ALWAYS is only used in special occasions (see section 2.3). In other occasions it must not be used together with a NEVER. Besides that, the number of possible numeric values is reduced. Only multiples of fifty between -5.001 and +10.001 will be used, which are 301 different values (instead of 64.001). This will have no influence on the solution, because numeric values below -1.000 or above 5.000 are seldom used and Elise will not calculate distinct match scores using the value 490 or 500.

Below the function $f(h_{space})$ is given and explained. This function is defined based on one tuning part and the size of the space depends on the number of properties in the related PRODUCT.DIC file.

equation 5.1. *search space function* $f(h_{space})$

$$f(h_{space}) = 2^{4 \cdot nProp} \cdot 301^{4 \cdot nProp} \cdot 3^{2 \cdot nProp}$$

with *nProp* equal to the number of properties in the PRODUCT.DIC file

The first part of the function is $2^{4 \cdot nProp}$. This part represents all the possible values for the WEIght, namely the possible values NEVER and a numeric value for the noobject, novalue, nomatch and match. The second part is $301^{4 \cdot nProp}$. This represents the possible numeric values related to the integer value of the WEIght attributes. The last part is $3^{2 \cdot nProp}$. This part represents all the possible values for the TYPe, namely the possible values EXClusive, REUse and OR for the RepeatingGroupType and OR, AND and INTersection for the MultiValueType.

The search spaces of the tuning problem are large but limited. When more properties must be tuned the space will increase with $2^4 \cdot 301^4 \cdot 3^2$ ($\approx 1{,}18 \cdot 10^{12}$) per extra property. The size calculated with the search space function is the maximum size of the space, but there is a way to decrease the actual size. Namely when substrings are disabled less string variations are possible because less parameter must be learned. Further, when the population is initialized based on analytical learning, the algorithm is already guided to an optimal parameter set and doesn't have to search the whole space.

5.5. *Fitness Function*

The fitness function calculates a fitness score for each string in the actual population. This score represents the fitness of the string. A high fitness means good match results while a low fitness means less match results. The fitness score also brings the possibility to compare the strings in terms of fitness. For example when string *X* has a higher fitness than string *Y*, the match results of string *X* will be better so string *X* represents a more optimal parameter set than string *Y*. Further the operators use the fitness scores to determine the probability of reproduction of each string. This will be discussed in section 5.7. The fitness function will be the same for the vacancy match score and for the resume match score and it will be expounded below.

To determine whether a match score is a good match score or not three factors are of importance when tuning by hand. Those are the differences between the target match scores and the actual match scores, whether the actual match scores fall within a predetermined range from the target match scores, and whether the matches are calculated correctly. Those quantities are of importance because they give a good picture of the match performance of Elise with the tuned parameter set. When the differences are small, the match scores fall within the predetermined range and the matches are calculated correctly, the domain is tuned well.

Those quantities will also serve as a base for the fitness function. Of each quantity an error function will be given and the fitness function will be built from those errors. The first error is the average difference between the target match scores and the actual match scores, which shows whether the actual scores come close to the target scores. This error must be small because a low deviation is related to a good performance. The second error is the percentage of actual match scores that don't fall within a predetermined range from the target match scores. This error must also be small because a sufficient amount of the matches must reach the target scores. Both are calculated because a small average deviation can be obtained by relatively small and large deviations whereas too much actual match scores can fall outside the acceptable range that means the performance of Elise is not sufficient. So the range error is needed besides the average error. The average error is also needed besides the range error because it must be able to compare the fitness scores of the strings with each other. The range error is a discrete error, which is not able to show small differences between the performances of Elise (which results in equal fitness scores) while the average error is continuous and can show (small) differences between the performances. The last error represents the percentage of matches that are not correctly calculated as a match or a nomatch. This third error is of importance because most matches must be calculated correctly.

For all three quantities an error can be calculated based on the difference between the target and the actual output of each match. The fitness function will use those errors to calculate the fitness score of each string. When a score is high, the performance of Elise is high (high fitness), and when a score is low, the performance is low (low fitness). This means that the fitness function must be maximized and therefore the three errors should be minimized. Below the error functions and the fitness function are given and explained.

equation 5.2. ε_{score}

$$\varepsilon_{score} = \frac{\sum_{i=1}^{n} \left| ActualMatchScore_i - TargetMatchScore_i \right|}{n}$$

with n the number of target matches

In equation 5.2 the average error between the *ActualMatchScore* (calculated by Elise), and the *TargetMatchScore* (given by the client) is calculated for each target match. This division is made to make this error function independent of the size of the learning set. ε_{score} reflects the average match error of each string in the population.

equation 5.3. ε_{range} and $\varepsilon_{\%range}$

$$\varepsilon_{range[i]} = \varepsilon_{range[i-1]}, \quad if\ TargetMatchScore_i - max\_err\_allow\% \leq ActualMatchScore_i \leq TargetMatchScore_i + max\_err\_allow\%$$

$$= \varepsilon_{range[i-1]} + 1, \quad otherwise$$

$$\varepsilon_{\%range} = \frac{\varepsilon_{range}}{n} \cdot 100$$

with n the number of target matches and i the ith string

Equation 5.3 calculates the percentage of matches that don't lie within the allowed range of *TargetMatchScore – max_err_allow%* and *TargetMatchScore + max_err_allow%*. When the *ActualMatchScore* lies within the range it is considered a good match, but when it lies outside this range, the match doesn't approximate the target enough, so it is considered as an error. *max_err_allow* will be set to 10% and when the range appears to be too wide it will be reduced. It is set to a relatively high percentage so the performance of the learning algorithm can be tested without the possible constraint of a range, which is too narrow.

equation 5.4. $\varepsilon_{matches}$, $\varepsilon_{nomatches}$, $\varepsilon_{\%matches}$, $\varepsilon_{\%nomatches}$, $\varepsilon_{\%totalmatches}$

$$\varepsilon_{matches\,[i]} = \varepsilon_{matches\,[i-1]} + 1, \quad if\ ActualMatchScore_i < Threshold \leq TargetMatchScore_i$$

$$= \varepsilon_{matches\,[i-1]}, \quad otherwise$$

$$\varepsilon_{nomatches\,[i]} = \varepsilon_{nomatches\,[i-1]} + 1, \quad if\ TargetMatchScore_i < Threshold \leq ActualMatchScore_i$$

$$= \varepsilon_{nomatches\,[i-1]}, \quad otherwise$$

$$\varepsilon_{\%matches} = \frac{\varepsilon_{matches}}{n} \cdot 100$$

$$\varepsilon_{\%nomatches} = \frac{\varepsilon_{nomatches}}{n} \cdot 100$$

$$\varepsilon_{\%totalmatches} = \frac{\varepsilon_{matches} + \varepsilon_{nomatches}}{n} \cdot 100$$

with n the number of target matches and i the ith string

In equation 5.4 five errors are calculated. $\varepsilon_{matches}$ represents the number of match errors. When the target is a match (*TargetMatchScore* ≥ *Threshold*) and the actual score a nomatch (*ActualMatchScore* < *Threshold*), it is considered as an error. $\varepsilon_{nomatches}$ represents the number of nomatch errors. When the actual match is a match (*ActualMatchScore* ≥ *Threshold*) but should be a nomatch (*TargetMatchScore* < *Threshold*), it is considered as an error. The last error is $\varepsilon_{\%totalmatches}$. This is the percentage of the total number of match and nomatch errors. This percentage is calculated over the total sum because the difference between the errors is of no importance to the fitness function. $\varepsilon_{\%matches}$ and $\varepsilon_{\%nomatches}$ are the error percentages of respectively the matches and the nomatches. Those are calculated separately because these are needed by the termination criteria (see section 5.6).

equation 5.5. *fitness function f(h)*

$$f(h) = \alpha_{score} \cdot (100 - \varepsilon_{score}) + \alpha_{range} \cdot (100 - \varepsilon_{\%range}) + \alpha_{totalmatches} \cdot (100 - \varepsilon_{\%totalmatches})$$

The fitness function *f(h)* is given in equation 5.5. This function is built from ε_{score}, $\varepsilon_{\%range}$ and $\varepsilon_{\%totalmatches}$. First the errors are subtracted from 100 because the errors should be minimized and the fitness score should be maximized. Second they are multiplied by respectively α_{score}, α_{range} and $\alpha_{totalmatches}$. The alpha factors express the importance of the related error in the fitness function. Those factors will initially be set to one because the importance of all errors is considered the same. When it appears the fitness scores don't express the performance of Elise well enough, changes to the alpha weights must be made. In that case α_{score} will be increased related to the other alphas because ε_{score} is the error that reflects the performance of Elise the best and therefore must be given higher weight when necessary.

When tuning by hand the attention given to the different errors differs during time. In the beginning it's important to calculate matches as matches and nomatches as nomatches. Later it becomes more important to reach the correct percentages for all matches. So the process is apparently different from the process of determining the fitness score because the fitness score considers the importance of all errors throughout the process. When the importance of the average deviation and errors would vary while learning, the algorithm would learn different targets and the process would be extended because then some parameter setting would have a good fitness score at training cycle *t* but doesn't have to have a good fitness at training cycle *t+10*.

5.6. Termination Criteria

The termination criteria are criteria that must be met to let the algorithm terminate. When the criteria are met using the training cases it means that Elise performs well enough with the parameter setting of the specific string. The performance will be checked with the test cases that must also meet the criteria using the specific parameter setting. When both type of cases have met the termination criteria, the algorithm has learnt the problem and stops.

In the previous section three factors are described, and of each factor an error function is defined. The importance of those errors is already described related to the fitness function, but their importance also counts related to the termination criteria. So the criteria are related to those errors and are defined in equation 5.7. Before those criteria will be checked, a string must have a higher fitness score than the minimum fitness allowed (see equation 5.6), otherwise the termination criteria can't be met.

equation 5.6. *min_fitness_allow*

$$min\_fitness\_allow = \alpha_{score} \cdot max\_err\_allow + \alpha_{range} \cdot max\_err\_range + \alpha_{totalmatches} \cdot max\_err\_matches$$

equation 5.7. *termination criteria*

$$\varepsilon_{score} \leq max\_err\_allow$$

$$\varepsilon_{\%range} \leq max\_err\_range$$

$$\varepsilon_{\%matches} \leq max\_err\_matches$$

$$\varepsilon_{\%nomatches} \leq max\_err\_nomatches$$

ε_{score} is calculated in equation 5.2, $\varepsilon_{\%range}$ is calculated in equation 5.3 and $\varepsilon_{\%matches}$ and $\varepsilon_{\%nomatches}$ are both calculated in equation 5.4. All errors must be smaller than the related *max_err* that defines the desired quality of the parameter set. *max_err_allow* has the same value as the allowed range for ε_{range}, namely 10%. *max_err_range*, *max_err_matches* and *max_err_nomatches* are initially set to 5% (together 10%). Those values are chosen wide, so the target values don't have to be approached closely, because the algorithm will be tested first. When it learns well the errors will be set smaller.

It is important that each individual error is smaller than the related *max_err*. Therefore the fitness score is not sufficient as the termination criterion because it is expressed as an average error and the three errors can neutralize each other. For example when $\varepsilon_{\%totalmatches}$ is 0 ($\varepsilon_{\%matches}$ is 0 and $\varepsilon_{\%nomatches}$ is 0), ε_{score} is 11 and $\varepsilon_{\%range}$ is 7, the fitness score is 282 but the individual part ε_{score} is too high. Further the string with the highest fitness is the most optimal string but it doesn't have to meet the criteria while another string with a lower fitness score can. For example when $\varepsilon_{\%totalmatches}$ is 5 ($\varepsilon_{\%matches}$ is 2 and $\varepsilon_{\%nomatches}$ is 3), ε_{score} is 8 and $\varepsilon_{\%range}$ is 7, the fitness score is 280. This score is lower compared to the other example but all errors meet the criteria. So besides the fitness score each error must be evaluated separately by the termination criteria. Furthermore, of the strings that meet the criteria, the string with the highest fitness score is the most optimal string.

It is also possible that the algorithm isn't able to learn the problem and the termination criteria will never be met. If this is the case the fitness will not improve anymore at a certain point and the maximal number of generations (see section 5.10) will be passed. When this happens, the algorithm must also terminate, but with the advice to check if the requirements and basic assumptions are met.

5.7. Operators

The purpose of the operators is to change the population based on the survival of the fittest principle. This means that the population will become fitter after each training cycle. The fittest strings are used to create new offspring so the next generation will contain fitter strings that result in better match scores. To determine the fitness of a string the fitness function is used (see section

5.5).

In section 4.3.2 the basic operators are shortly discussed. Those are the reproduction, crossover and mutation operator. The reproduction operator selects the (fittest) strings that must mate to create new offspring. When two strings are selected to mate the crossover function will switch parts of those strings to create two new strings. The mutation operator mutates parts of the new strings and the resulting strings will be inserted into the next generation. This is a random process and because it is based on the survival of the fittest principle each new generation will be fitter than the previous one. Besides this the operators must take care of the structure of the string (see section 5.3) so the encoded information can always be decoded. Below the operators will be discussed in greater detail.

5.7.1. Reproduction

Reproduction is the operator that takes care of the selection process. It selects pairs of strings that must mate to create offspring for the next generation. This selection is based on the fitness score. When the fitness score of a string is high, its chances of reproduction are higher than when the fitness score is low. The roulette wheel method (see 4.2.2.1.) is the original method to implement this selection process. A disadvantage of this method is that (relative) highly fit strings take over the population, which reduces the diversity of the population and the algorithm only explores a (sub optimal) part of the search space. This way it is not guaranteed the algorithm finds an optimal solution. This phenomenon is known as premature convergence. To avoid this phenomenon, Boltzmann selection is used. For advantages and disadvantages of the different selection methods see M. Mitchell (1996).

Besides selecting strings to mate, the fittest strings can also be copied to the next generation, which is the elitism method. Both operators are described below.

5.7.1.1. Boltzmann Selection

The Boltzmann selection method uses the simulated annealing principle. Instead of using the fitness score, it uses the expected value of a string. The expected value is the expected number of times a string will reproduce. The fitness score and the expected value are both related to the chance of reproduction of that string. To calculate the expected value, temperature T is used. Temperature T controls the value during the learning process. Therefore it is initially set high and is gradually lowered. This means that in the beginning of the learning process strings with a relatively low fitness have a higher change of reproduction than without using the simulated annealing principle and strings with a high fitness have a relatively lower change of reproduction. When the temperature lowers, the difference between the expected values of strings with a low fitness and with a high fitness increases. This way the algorithm will find the optimal solution in the best part of the search space (M. Mitchell, 1996).

The temperature will change according to a preset schedule. The temperature T will initially be set to 105. The temperature will be lowered with 5 each time an extra 5% generations of the maximum number of generations is passed, until it is equal to 5. The equation to calculate the expected value is given below.

equation 5.8. *exp_val*

$$exp\_val(i,t) = \frac{e^{f(i)/T}}{\left\langle e^{f(i)/T} \right\rangle_t}$$

with T the temperature, t time t and i the string

When the expected value of every string is calculated, the roulette wheel will finish the selection process. Strings with a higher expected value will have a higher probability of reproduction than strings with a lower expected value because they get a larger part of the roulette wheel. The size of that part is related to its expected value that is translated into a percentage. When each string has a part of the wheel, the wheel is spun n times (n is the number of strings in each generation). The spinning result will be a randomly generated number between zero and one hundred that is related to a part of the wheel. The string related to that part is selected for reproduction. The spinning will be repeated to select pairs of strings that must mate until the number of strings in the next generation is equal to the population size.

5.7.1.2. Elitism

Besides Boltzmann selection elitism will also be used for reproduction. Elitism is introduced by K.A. de Jong in 1975 and prevents the algorithm from losing the best strings of a population. Therefore the fittest strings will be copied to the new population without crossover and mutation changing the strings. This increases the performance of the algorithm because the best strings are kept and not lost by accident because they were not selected to mate or because the resulting string after mating has a lower quality than the original string.

In order to let the algorithm learn so the population becomes fitter after each training cycle, there must be variations between the strings of each population. Therefore the elitism parameter is initially set low to 4%, so only 4 percent of the new generation consists of copied strings and 96 percent of newly created strings.

5.7.2. Crossover

There are three types of crossover that can be used for binary and value encoding. The first is single point crossover where a single point in the string is randomly selected and the bits or values after that point are exchanged between the strings. The second is two-point crossover. Here two points are randomly selected and the bits or values between those points are exchanged. The last one is uniform crossover where bits or values are exchanged randomly between the strings. Two-point crossover and uniform crossover are used most commonly. Using uniform crossover can be highly disruptive of any possible structures within a string where two-point crossover is not (M. Mitchell, 1996). Therefore, two-point crossover will be used as operator for both the binary encoded string part and the value encoded string part.

For the binary encoded string part it is important that the structure of the substrings of three bits (see section 5.3 on representation) is maintained. This means that before and after crossover each substring must contain one active bit (1) and two inactive bits (0). This can be done exchanging substrings instead of single bits. Together with each substring the related numeric value (of a WEIght substring) will be exchanged. This is done because the numeric value is strongly related to the substring (when the integer bit is active) so the fitness score is dependent on the combination of those two.

Now that the crossover operator is defined, the crossover probability must be determined. This probability is the chance that the new generation will be generated by crossover. When the probability is 100% the new offspring will be generated completely by crossover and when it is 0% the new offspring will exist of copies of the selected strings. A study of genetic algorithms in function optimization (K.A. de Jong, 1975) suggested that good genetic algorithm performance requires the choice of a high crossover probability and a low mutation probability.

Elitism is already used to copy the fittest 4 percent of the population to the next generation, so the

crossover probability can be set high and is initially set to 95%. This means that each selected string pair has 95 percent chance on exchanging bits and 5 percent chance on being copied before the mutation operator is called.

5.7.3. Mutation

When crossover has taken place, the mutation operator will change the newly created strings. Mutation is different for bits and values. For bit string encoding mutation inverts bits. For value encoding mutation will add or subtract a (small) number to or from the original number. Both can be done randomly for one or more positions in a string.

Mutation is necessary to prevent bitstrings from losses (see section 4.3.2.3). For value encoding it is important that the values change to learn the correct values. Using only crossover just changes combinations of the initial values, so mutation is needed to change the values. Therefore two mutation operators are used, the bitstring mutation operator and the numeric value mutation operator.

For the bitstring mutation operator it is important that the structure of the substrings of three bits is maintained. This means that before and after mutation each substring must contain one active bit (1) and two inactive bits (0). So when a bit is inverted the whole substring must be checked and bits must be mutated when necessary. For example when the substring 001 is mutated to 101, than the last 1 must be inverted also so the substring becomes 100. When a substring is disabled (%%%), mutation is not possible.

The numeric value mutation operator will use a kind of simulated annealing. It will not use a temperature that is lowered during the learning process, but it will lower the maximum possible change of the numeric value. The number that will be added or subtracted to or from the numeric value can be high in the beginning of the learning process and will be small at the end. The following schedule of possible changes will be used: 50 till 1000 at the beginning, when 25 percent of the maximum number of generations is passed it becomes 50 till 750, 50 till 500 when 50 percent is passed and 50 till 250 at the end of the learning process (when 75 percent is passed).

At last the mutation probabilities must be determined, one for the bit string mutation and one for the value mutation. A probability of 100% means a total different string is passed to the new generation, and one of 0% means the original string is passed. As mentioned in the previous section the mutation probability should be low, but this only counts for the bit strings because values should be mutated more often to differ from the initial values (see above). So the bit mutation probability is set initially to 0,1% and the value mutation probability is initially set to 25%. This means that each bit has 0,1 percent chance on mutation and each value 25 percent chance.

5.8. Population

The population of a genetic algorithm consists of strings. During learning this population evolves and each generation becomes fitter. The population size and the initialization of the population are two important aspects, which influence the performance of the algorithm. Both aspects are discussed in the next two subsections.

5.8.1. Population Size

The population size is the number of strings in the population (in one generation). This size is important for the performance of the algorithm. When there are too few strings (compared to an adequate population size) it takes longer to search the whole space and learning is slowed down. When there are too many strings, the solution will not be learned faster because more

(unnecessary) strings must be evaluated per training cycle. So the whole learning process is not improved (G. Harik, D.E. Goldberg, E. Cantú-Paz, L. Miller, 1999).

However, it is difficult to determine the adequate population size. In the beginning a population size of 50 strings up to 100 strings was considered a good size, but later it became clear that the size depended on the problem that must be learned. There are only few studies on this subject and those studies deal only with binary encoded strings. So they cannot be used for the tuning problem. Therefore, the factors used to determine the population size that can be related to the tuning problem are discussed below and will be used to give an estimation of the needed population size.

Two valid argumentations are given. First, problems with strings built from short building blocks are easier to solve than problems with strings built from long building blocks, because long building blocks are scarcer and the structures within strings with long building blocks are easily disrupted. Therefore, problems with strings built from short building blocks need smaller populations. Secondly, the size of the learning problem plays a role. When the problem size increases, the population size must also increase (G. Harik, D.E. Goldberg, E. Cantú-Paz, L. Miller, 1999).

In the tuning problem the building blocks are short. Besides that, it is impossible to disrupt them because the crossover operator takes care of the static structure of the substrings. So, the size of building blocks can be considered as one. Further, problems with more properties to learn need larger population sizes, because the search space becomes more complex for each extra property to learn.

So the population size of the tuning problem depends on the number of properties in the PRODUCT.DIC that must be learned and the number of building blocks in a one-property string. With equation 5.9 an estimation of the population size is given.

equation 5.9. *pop_size*

$$pop\_size = 5 \cdot nWEIghtBBs \cdot nProp + nOtherBBs \cdot nProp$$

with *nProp* the number of properties to learn, *nWEIghtBBs* equal to the number of WEIght building blocks in a one-property string (which is 4) and *nOtherBBs* equal to the number of other building blocks in a one-property string (which is equal to the 2 TYPe parameters)

The equation is divided into two parts. One part calculates the population size based on the number of WEIght parameters that must be learned. This part is given five times the weight of the second part, which gives the population size based on the number of parameters that must be learned that consists only of bit substrings. The population size is mainly determined by the number of WEIght parameters to be learned because the numeric values related to the WEIght substrings will be the hardest to learn. Often they will not be present in the initial population while the bit substrings will, as discussed in section 5.3. This size can be reduced with the number of fixed parameters multiplied by the related factor. The definitive population size can be compared to population sizes of different genetic algorithms that use value encoding, which differ from 50 strings (relative simple search space) up to 1000 strings (complex search space).

Note this is only an estimation. To determine the adequate population size, further research must be done.

5.8.2. Initial Population

In chapter 4 analytical learning is introduced. Analytical learning uses prior knowledge to increase the learning speed. One option to use prior knowledge is initializing the initial hypothesis, in the case of a genetic algorithm the initial population. This initialization can be done by hand and variations of this initial string can be made automatically. However, for a genetic algorithm it is of importance that there are substantial variations between the strings of the initial population and a lot of variations are allowed to be poor solutions. When initializing the whole population using variations there is a chance the variations are too small and the algorithm gets stuck in that small part of the search space. Therefore a part of the initial population can be randomly generated. Because it is not known which initial type will give the best learning results, different initializations will be compared during development to determine the best way to initialize the population. The different initialization options will be complete initializing using variations, a complete random initialization or a mix between the previous two. For variations of the initial string, each bitstring or numeric value has a chance of 25 percent to be changed. Each numeric value will be increased or decreased by minimal 50 and maximal 1000.

5.9. Control Parameters

In the previous sections all the control parameters are already discussed and defined. Those control parameters must be set up before learning starts. An overview of those parameters is given in table 5.2 on the next page. The domain independent parameters concern the fitness function, the termination criteria, the operators and other parameters that are respectively defined in section 5.5, 5.6, 5.7 and 5.7 and 5.8. Those control parameters are initially (and wide) and can be changed when it appears while learning that the values are not correct. The domain dependent control parameters depend on the number of properties of the PRODUCT.DIC file (except the threshold). The threshold is discussed in 5.5, the number of training and test cases in 5.2 and the population size equation is given in section 5.8.

Domain Independent Control Parameters	
Fitness Function f(h)	
max_err_allow	10%
α_{score}	1
$\alpha_{\%range}$	1
$\alpha_{\%totalmatches}$	1
Termination Criteria	
min_fitness_allow	calculate with equation 5.6
max_err_allow	10% (see above)
max_err_range	10%
max_err_matches	5%
max_err_nomatches	5%
Operators	
elitism	4%
pCrossBit (crossover probability control genes)	95%
pMutBit (mutation probability control genes)	0,1% (small: only to prevent loss)
pMutVal (mutation probability coefficient genes)	25% (large: numeric values must vary)
Other	
rnd_pop_init (number of random generated strings in initial population)	0%, 50%, 100% (test values)
variation probability of initial string	25% (with + or - 50 till 1000 of the numeric values)
temperature T_selection	105, gradually lowered with 5 each time an extra 5% generations of the maximum number of generations is passed, until T=5
schedule mutate_numeric_values	50 till 1000 (first 25% number of generations), 50 till 750 (next 25%), 50 till 500 (next 25%), 50 till 250 (last 25%)
Domain Dependent Control Parameters	
threshold	dependent on matching domain
nCases (number of training and test cases)	8 till 10 per property per training and test set
popsize (population size)	calculate with equation 5.9 (depends on number of properties in PRODUCT.DIC)

Table 5.2. *Control Parameters*

5.10. Technical Aspects

Besides the design of the genetic learning algorithm, the technical aspects are also of importance. In chapter 2 the matching tool is already introduced, namely Elise. The version that will be used is version Elise 4.00.007.

The programming language that will be used to implement the genetic algorithm is C++. An object-oriented language is chosen because several objects are needed (string, population, target

matches). Another pro of C++ is the speed.

Other technical factors are the complexity of the algorithm and the computation time. Those factors will be discussed in respectively subsection 5.10.1 and 5.10.2.

5.10.1. Complexity

Below an estimation of the complexity of the genetic algorithm is given. It's a setting out of the basic functions of the genetic algorithm. The calculations that must be done by Elise are not taken into account. For the following estimation, the book "Algorithmics" (D. Harel, 1992) is used.

Table 5.3 below shows the worst-case orders of the different functions of the genetic algorithm. Initializations, logs and tests are not taken in. The calculations are based on one generation and calculated for two different PRODUCT.DICs. In the first, five properties must be learned and in the second, ten properties must be learned. Based on those properties the population size, and the number of cases are determined, which grow linearly.

All the functions stay in polynomial time except one, namely "check if new string is not in new population yet", which is near exponential time. With few properties to learn, this gives no problem. However, when more properties must be learned, the algorithm needs disproportionately more time. Therefore, it is an option to remove this function or fix the maximum number of generations related to the maximum amount of time the algorithm is allowed to learn.

	5 Properties (N)	**10 Properties (N)**	**Order of Algorithm**
population size	105	210	
number of cases	+/- 50	+/- 100	
steps per generation:			
calculate fitness	105 · 50 · 15	210 · 100 · 15	$O(3150N^2)$
calculate expected value	105 · 2	210 · 2	O(42N)
check termination criteria	105 · 5	210 · 5	O(105N)
elitism	105 · (4% of 105)	210 · (4% of 210)	$O(18N^2)$
selection	105 · 4	210 · 4	O(84N)
crossover	(105/2) · (15+4)	(210/2) · (15+4)	O(200N)
bit mutation	105 · 5 · 15	210 · 10 · 15	$O(315N^2)$
value mutation	105 · 4 · 2	210 · 4 · 2	O(168N)
check if new string is not in new population yet*	(105-1)!	(210-1)!	O((21N-1)!)
compare strings	5 · (15+4)	10 · (15+4)	O(19N)
replace population	105 · ((5 · 15+4)+2)	210 · ((10 · 15+4)+2)	O(441N)
total	(105-1)!	(210-1)!	O((21N-1)!)

table 5.3. *Example of Computation Complexity*

*) equal strings in one population do occur in every run, but not very often, so with many properties to learn, this check can be left out, because equal strings will appear even less with longer strings.

5.10.2. Computation Time

The computation time is related to several factors. Those are the population size, the maximum number of generations and the number of training cases. The population size and the number of

cases are already discussed, the maximum number of generations is not. The maximum number of generations is the number of generations after which the genetic algorithm will stop learning, also when the termination criteria aren't met yet.

Within the genetic algorithm domain it is common to relate the maximum number of generations to the time the algorithm is allowed to learn. Equation 5.10 calculates the time T based on the previously mentioned factors. This equation is based on the model of total time T (Fitzpatrick and Grefenstette, 1988). In equation 5.11 it is rewritten to calculate the maximum number of generations.

equation 5.10. *time T*

$$T = \alpha_1 + \alpha_2 + pop\_size \cdot \beta_1 + nCases \cdot \beta_2 + \left(pop\_size \cdot \beta_3 + nCases \cdot \beta_4\right) \cdot max\_gen$$

with α_1 the initial amount of costs of the genetic algorithm, α_2 the initial amount of costs of Elise, β_1 the costs of one generation minus the string evaluations, β_2 the costs of one Elise match call minus the single match calculations, β_3 the costs of one string evaluation and β_4 the costs of a single match calculation by Elise, where β_1 and β_2 grow quadratic when more properties must be learned (see table 5.3)

equation 5.11. *max_gen*

$$max\_gen = \frac{T - \alpha_1 - \alpha_2 - pop\_size \cdot \beta_1 - nCases \cdot \beta_2}{pop\_size \cdot \beta_3 + nCases \cdot \beta_4}$$

idem

6. Implementation

The implementation of the genetic algorithm is based on the design given in chapter 5. However some changes have been made. Some of them are made because the time factor didn't allow implementing all the functions. The changes are discussed in section 6.1. Besides the changes, some extra functions are implemented to record the performance of the genetic algorithm to evaluate the performance of the algorithm and at the same time the performance of Elise with the parameter settings proposed by the algorithm. Those functions are discussed in section 6.2.

Further, the communication from the genetic algorithm to Elise is implemented. This communication is not optimal. Each time Elise must calculate match scores with a new parameter setting the DataDictionary is reloaded. Then the matches are executed using a batch file. Both are time consuming processes. It is much quicker to use a communication tool that would communicate more directly to Elise without reloading the DataDictionary, but due to the scarce time this was not possible. However, the pseudo-code of this tool is given in appendix F. The complete programming code of the genetic algorithm is given in appendix E.

6.1. Design Changes

During the implementation phase some adjustments have been made to different parts of the design of the algorithm. Those adjustments are discussed below in the different subsections.

6.1.1. Prior Knowledge

Based on the experience with tuning by hand it is known that NEVERs, zero values and positive values often occur as WEIght attribute values in the final PRODUCT.DIC while negative values rarely occur. This knowledge is used to generate the initial population. It is chosen to give the same probability to a NEVER and a numeric value. When a numeric value is generated, positive weights and the zero weight have the same chance to be generated, and negative weights have a small chance to be generated. When a NEVER is generated, the related numeric value is set to zero. This value is closest related to the NEVER because the NOMATCH attribute is set to a NEVER or a zero most of the time.

6.1.2. Fixed Parameters

Due to scarce time, the support for fixed parameters isn't implemented.

6.1.3. RepeatingGroupType

It turned out that the DataDictionary must be reloaded when the RepeatingGroupType must be changed. This is a time consuming process, which will lengthen the learning process in an unacceptable way. However, most of the time it is known whether EXClusive, REUse or OR must be used as value of this TYPe. Therefore, the genetic algorithm does not learn the RepeatingGroupType (as long as this technical constraint is not solved).

6.1.4. Population

Strings can occur only once in one generation. This is decided to guarantee the diversity of the population. When (fit) strings were allowed to occur twice or more times in one generation, they can take over a large part of the population, which can influence the learning process in a negative way. First the best 4% strings will be copied to the new population before new offspring is created. When it appears that a newly created string is already in the new population, the string is deleted and a new one is produced. However, when the number of properties that must be learned becomes too large, it is an option to allow equal strings in one generation to reduce the

computation time (see subsection 5.10.1).

6.1.5. Boltzmann Selection

During the first tests of the genetic algorithm it appeared that during the last generations the expected value percentages of the many strings became extremely small and the expected value percentages of a few strings summed almost to one hundred percent. This extreme difference is unwanted and can be attributed to a minimal temperature that could drop too low. Besides that, the log files showed that the initial temperature was also too low. The fitness percentage and expected value percentage of a string in the first generations did not differ significantly. Therefore the preset schedule of the temperature is changed. The temperature T will initially be set to 205. The temperature will be lowered with 10 each time an extra 5% generations of the maximum number of generations is passed, until it is equal to 15.

6.1.6. Mutation

When a NEVER is active on a WEIght attribute, the related numeric value will not be changed because the NEVER is related to the zero (see subsection 6.1.1). When it would be possible to change the numeric value, most zero values, which are related to the NEVER values, would disappear. In consequence, it probably will be harder for the algorithm to find an optimal solution because zero values occur often in the final parameter setting (when tuning by hand).

6.1.7. Termination Criteria and Test Cases

It is possible that the termination criteria are met using the training data, but not using the test data. According to figure 5.1 the algorithm terminates without an optimal parameter setting, but it is better to let the algorithm continue learning. Namely, the parameter setting of the string can be near optimal. The fitness of the strings probably will improve within the next few generations, so the algorithm will terminate with an optimal parameter setting. When this is not the case, the algorithm will terminate after the maximal number of generations with the recommendation to check the consistency of the cases, the matrices and the design of the matching application. Therefore the algorithm continues learning and only terminates when an optimal string is found with the training and test cases or the maximum number of generations is exceeded.

6.2. Extra Functionality

During the learning process, several log files were kept up. The first one (matchScores.txt) logs the actual match results, the difference compared to the target match score, the errors and fitness score per string per generation. The second file (results.txt) records all the strings that are fit enough considered by the termination criteria. The string number, the generation number and the fitness score are logged. It is also logged whether or not the problem is learned when the maximum number of generations is exceeded. The most fit string up to then and its parameter values of all properties are written to the results file. The third file (resultsPop.txt) records the fitness scores, the expected values and the related percentages of each string of each generation. This file is used to control the temperature schedule of the selection operator. The last log file logs all the populations. Of each generation all the encoded strings are written to the logPop.txt file together with the related fitness score and the parents (except for the initial population). Besides that of each generation the number of crossing overs, the number of bitstring mutations and the number of value string mutations (except for the last generation) are recorded. The maximum fitness score, the minimum fitness score, the sum of the fitness scores, the average fitness score and the maximum fitness score over all generations are recorded as well.

The maximum fitness score of the current generation is kept (online performance analysis) and the maximum fitness score over all generations is kept (offline performance analysis). However, using elitism the strings with the highest fitness are always copied to the next generation, so the

highest fitness score of the current generation is always the highest fitness over all generations.

7. Tests and Results

Now the genetic algorithm is implemented, its performance must be tested to see if the learning algorithm can be used to tune the PRODUCT.DIC files of Elise. As mentioned in the introduction, this will be done with an existing project of Bolesian. This project is described below. With this project, the settings of the control parameters are checked during a few try-out runs and changed when it appears necessary. After that, three experiments are run and the results are interpreted.

7.1. The Real Life Case

The real life case is the matching application of a temping agency that is tuned by hand. Based on training cases the match scores are evaluated and the parameters are set. With those parameter settings, acceptable match scores are achieved. Those match scores are used as target match scores for the genetic algorithm. Besides the original training cases, nomatch cases with target scores are added to the training set, which result in a training set of 68 cases. This is done because only match cases were present (see subsection 5.2). All the used training cases are given in appendix C. The genetic algorithm compares the actual match scores of the training cases (the ones Elise calculates) to the target match scores. Based on the differences, the fitness score is calculated and the performance of the algorithm and Elise can be evaluated.

The temping agency is a relatively simple project because of the limited amount of matching properties and the fact that the matches are calculated from just one side (the vacancy side), not from two sides. Therefore, the algorithm is tested for the vacancy PRODUCT.DIC file only. This PRODUCT.DIC contains ten properties of which five properties must be learned. Those are the properties with at least one WEIght attribute with a numeric value different from zero (which means different from WEI(0, 0, 0, 0) or WEI(0, 0, NEVER, 0)), because those properties have influence on the match scores. Properties that don't have to be learned are properties like ID, name, postal code, which have no influence on the match scores.

For only one of those five properties the parameter TYPe is defined. Of this parameter, only the attribute MultiValueType is defined, because just multi values are allowed and multi instances are not allowed. Of the other properties, neither multi values nor multi instances are allowed, so defining the TYPe parameters for those properties is of no use. Therefore, only the MultiValueType of one property is learned, the other TYPe parameters are left out.

The PRODUCT.DIC files of the temping agency are given in appendix B.

7.2. Control Parameters

During the first try-out runs of the genetic algorithm, the control parameter settings of table 5.2 are used. However, it became clear that the values of some control parameters had to be changed. Those parameters are the alpha-range, the maximum error-score, -range, -match and -nomatch and the Boltzmann selection temperature schedule. The changes are explained below.

7.2.1. Alpha-Range

During the try-out runs, it became clear that the fitness score didn't reflect the performance of Elise well enough. Generally, the fitness score was too high when the performance was moderate. For high fitness scores, the error-score, -match and -nomatch were relatively low, but the error-range was often higher (between 10 and 14%). Together, this resulted is a fitness score higher than the minimum fitness allowed to meet the termination criteria. To correct the fitness score, it

is decided to give the alpha-range a higher weight (instead of the alpha-score as mentioned in subsection 5.5) and the alpha-range is changed from 1 to 2. This way the performance of Elise will be expressed more accurately.

7.2.2. Maximum Errors

Besides the alpha-range, the maximum errors are modified. During the first runs it appeared that the fitness scores became relatively high during the first generations, although target match scores weren't approached very well. To be able to relate the fitness score to the performance of Elise more accurately, the maximum errors are set tighter. Therefore the maximum error-score is set to 5 instead of 10, the error-range is also set to 5 instead of 10 and the error-match and -nomatch are changed from 5 to 2.

7.2.3. Boltzmann Selection

During the try-out runs, the match results showed that the percentages of the expected values didn't follow the schedule as they were expected to. At the beginning of a run (the first generations) the percentages of the expected values were almost equal to the percentages of the fitness scores. However, the expected value percentages were supposed to be closer to each other than the fitness score percentages. Therefore, the starting temperature is increased to 295 (instead of 205). At the end of a run (the last generations) the percentages didn't differ much because the fitness scores and expected values all were high. However, at the end of a run the algorithm must be able to express the difference between a fitness score of 280 and 295. Therefore, the final temperature is lowered to 10 (instead of 15). It is allowed to lower the final temperature, because the fitness scores don't differ much at the end of the learning process, so the expected values won't become too small.

Those three changes result in a better reflection of the fitness score related to the performance of Elise and better expected values during a run. To summarize the changes, the definite values of the control parameters are given below. Those will be the same for all three experiments.

```
Static Control Parameters:
maximum error allowed: 5
maximum error range: 5
maximum error matches: 2
maximum error nomatches: 2
alpha score: 1
alpha range: 2
alpha total matches: 1
chance on variations on initial string: 25
chance on elitism: 4
crossover probability: 95
mutation probability numeric values: 25
mutation probability bit string: 0.1

Schedule Temperature Boltzmann Selection:
initial temperature: 295
gradually lowered with 15 each time an extra 5% generations of the
maximum number of generations is passed
final temperature: 10
```

Besides the control parameters, the genetic algorithm uses variables. Some of them are already discussed in subsection 7.1, the others are discussed in the next paragraph.

The threshold to separate the matches from the nomatches is set to 40%. The number of generations is set to 30 and the population size is also set to 30. Due to the used communication

between the genetic algorithm and Elise it was not possible to test the performance of the algorithm in full during the limited time. Therefore it is decided to use a smaller population size and fewer generations than required. For this reason, the test cases to evaluate the learned parameter set are left out too. The used variables are given below and will also be equal for all three experiments.

```
Static variables over all experiments:
number of generations: 30
population size: 30
threshold: 40
number of cases: 68
number of properties to learn: 5
```

The variable that isn't mentioned yet, is the percentage of randomly generated strings in the initial population, as discussed in subsection 5.8.2. This percentage will differ for the three experiments. Experiment 1 will use 50% random generations and 50% variations, experiment 2 will use 100% variations and experiment 3 will use 100% random generations. Those experiments are discussed in the next section and the complete test results of the experiments are given in appendix D.

7.3. Experiments

7.3.1. Experiment 1

Experiment 1 is executed with the settings described in the previous sections. The results of the experiment are shown in figure 7.1.

The average fitness quickly grows towards a fitness score of 330 where the minimum allowed fitness is 383. At the end of the run the average fitness approaches the maximum attainable fitness of 400. The maximum fitness grows from 332 in the first generation towards almost 400. The first time the algorithm meets the termination criteria is in generation 13 with a fitness score of 398,464. In the last generation the maximum fitness is 399,379. Those results show that the genetic algorithm is able to approach the target match scores very accurate with the given settings.

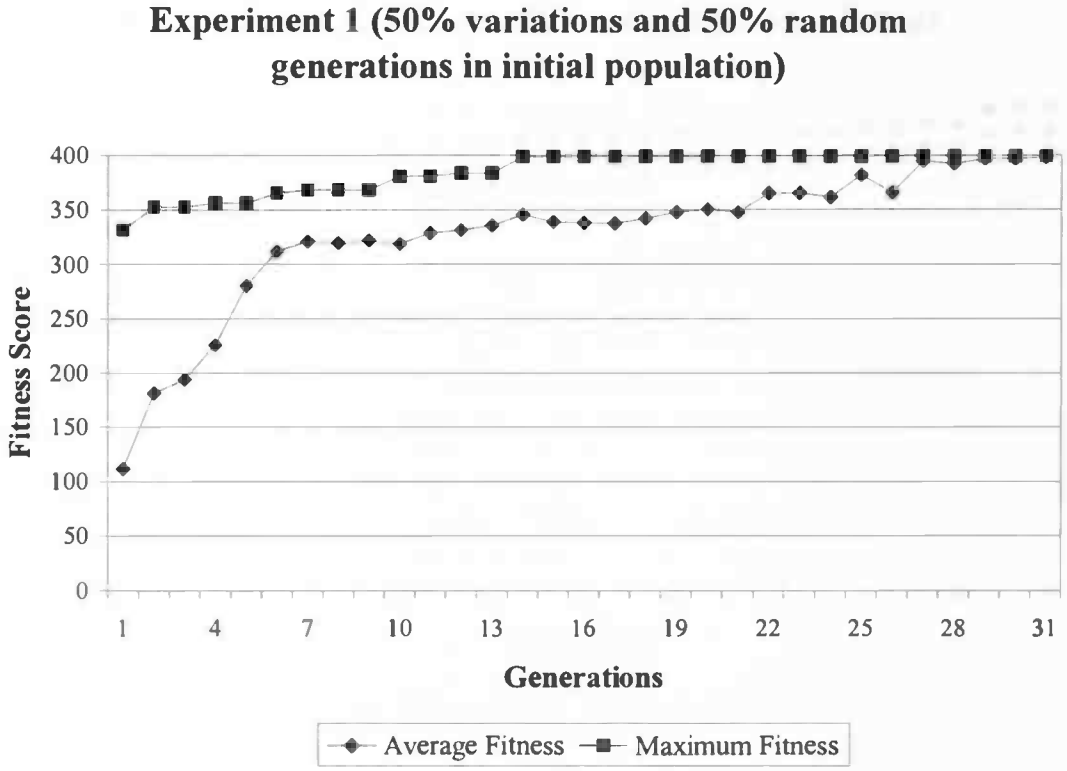


Figure 7.1. *Fitness Scores Experiment 1*

7.3.2. Experiment 2

Experiment 2 is also executed with the settings described in the previous sections. The results of this experiment are shown in figure 7.2.

The average fitness quickly grows towards a fitness score of 340 where the minimum allowed fitness is 383. At the end of the run the average fitness approaches a fitness of 380. The maximum fitness grows from 345 in the first generation towards almost 400. The first time the algorithm meets the termination criteria is in generation 28 with a fitness score of 395,581. In the last generation the maximum fitness is 398,976. Those results show that the genetic algorithm is able to approach the target match scores with the given settings.

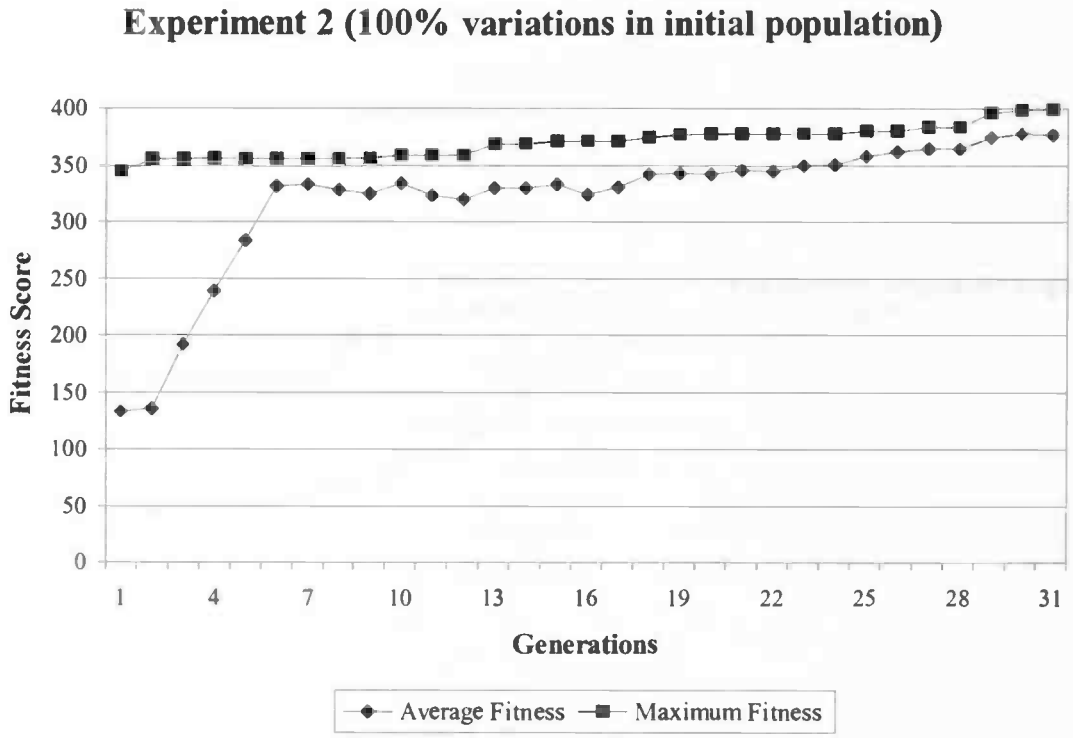


Figure 7.2. *Fitness Scores Experiment 2*

7.3.3. Experiment 3

The last experiment is experiment 3 that is executed with the same settings as the previous experiments except for the initial population, which is randomly generated. The results are shown in figure 7.3.

The average fitness slowly grows towards a fitness score of 220 where the minimum allowed fitness is 383. At the end of the run the average fitness approaches a fitness of 281. The maximum fitness grows from 190 in the first generation towards 283. The algorithm doesn't meet the termination criteria during this run, which means that the genetic algorithm isn't able to approach the target match scores with the given settings.

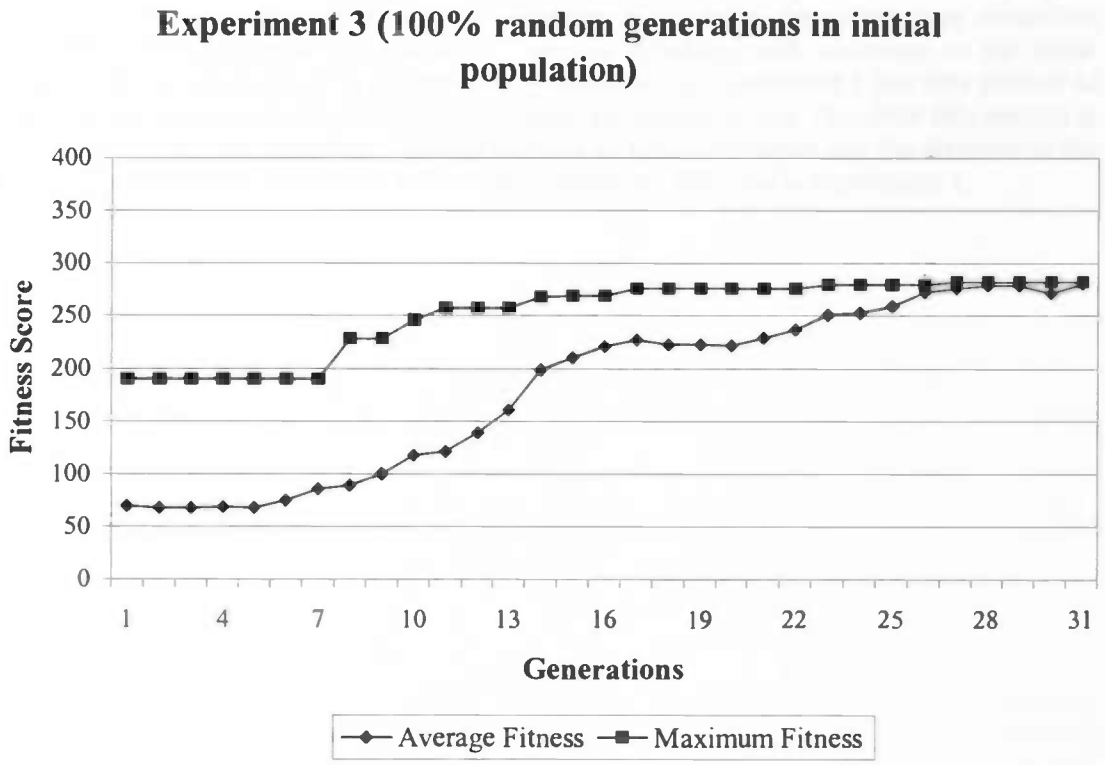


Figure 7.3. *Fitness Scores Experiment 3*

Besides the positive overall results, more results could be extracted from the log files. Those results concern the final parameter settings. All the WEIght values of the final parameter settings are equal to NEVER or a numeric value different from zero (see appendix D). This means that the algorithm is not able to tune zero WEIght values at this moment. A solution to this problem is given in chapter 8. Another remark must be made about the NOOBJECT and NOVALUE WEIght values. The differences between those values are large between the three experiments, so the values aren't useful or reliable. This can be attributed to the domain. The design doesn't allow NOOBJECT and NOVALUE matches to occur, so the strange values cannot influence the match scores. In such designs, those WEIghts should be fixed to zero before learning starts. However, the strange values can also be attributed to the training set, which only contains match and nomatch cases and no novalue and noobject cases. When NOOBJECT and NOVALUE matches can occur, those cases must be added to the training set to obtain reliable results for the corresponding WEIght values.

Further, it was expected that runs with initial populations with variations on the initial provided string would be guided more quickly towards an optimal solution than populations that were completely randomly generated. This is validated by the three experiments. Experiment 1 and 2 use variations and grow quickly towards an optimal solution. Conversely, the algorithm is not able to approach the target matches with the initialization of experiment 3. To obtain better results with a randomly generated initial population, the algorithm probably needs a larger population size and more generations. However, it was not possible to test this option, due to the slowness of the communication between Elise and the genetic algorithm.

Although the test results of experiment 1 and 2 both are positive, experiment 1 shows slightly better results. This can be attributed to the diversity of the initial population (see subsection 5.8.2). The initial population of experiment 2 is only initialized with variations on the initial parameter setting, which result in a low diversity. However, in experiment 1 just fifty percent of the initial population exists of variations on an initial parameter setting. The other fifty percent is randomly generated, the algorithm is guided towards an optimal solution and the diversity of the population is guaranteed. Therefore, better match scores are achieved in experiment 1.

8. Discussion

The test results in the previous chapter are mainly positive results. The diagrams of experiment 1 and 2 show hopeful prospects for future work. The results will be discussed extensively in this chapter. Therefore, they will be related to the definition introduced in chapter 1 and the requirements in chapter 3. First, the definition of the introduction will be repeated. After that the different issues will be discussed. At last, recommendations for future work will be made in section 8.1.

Definition

In matching applications match scores are calculated between demand and supply. The match scores are calculated using different parameter settings for different criteria. Those parameters are tuned by hand.

The goal is to determine whether or not machine learning techniques can be of use in tuning those parameters automatically. If so, which machine learning algorithms are appropriate and under what conditions can they be used? Therefore it must be investigated what can be learned by an algorithm and what must be defined within the domain. The most appropriate algorithm must be implemented and tested with training data.

In chapter 4 three learning algorithms were investigated. Those were knowledge-based learning, genetic algorithms and decision tree learning. All algorithms were discussed in terms of generalizibility, robustness, chance on success, time profit, the possibility of learning extra parameters and the amount of needed training and test data. Only the genetic algorithm was valued positive on all those factors (see table 4.6). Therefore, a genetic algorithm is used to solve the tuning problem and it is implemented and tested. The global test results were already discussed in chapter 7 and below they will be discussed in terms of requirements related to the learning conditions.

- *For all the properties in the PRODUCT.DIC files the WEIghts must be learned.*
  The results show that learning a numeric value and the value NEVER doesn't give any problems in experiment 1 and 2 (see final settings in appendix D). However, it appeared that the value zero was more difficult to learn, because no learned parameter setting contains this WEIght value. This can be attributed to the value mutation operator, which changes the numeric values. Changing a numeric value into a zero has a lower probability than changing it into a non-zero value. However, it is important that a zero can be learned, because zero is a regular occurring value. A possible solution is to change the ALWAYS-bit (which is not used) of the WEIght bitstring into a zero-bit. When this bit is set active, the value for the related WEIght is zero.
  Besides the difficulty to learn a zero, the results show unwanted and unpredictable values for the noobject and novalue WEIghts in the learned parameter setting. This is because no cases were present in the training set that covered this type of matches. However, they must be present when those attributes must be learned. This increases the number of needed training examples per property to ten till twelve instead of eight till ten, because more cases and more differentiation between the cases is needed when more parameters must be learned.
- *The TYPes must be learned for all properties in the PRODUCT.DIC files.*
  The RepeatingGroupType is not learned (as discussed in 6.1.3) due to technical restrictions. This restriction will be solved, so it will be able to learn this TYPe. However, often the setting of this parameter can be set based on experience. In such cases it doesn't

have to be learned.
Experiment 1 and 2 show positive results related to the MultiValueType (see final settings in appendix D), so the algorithm is able to learn this TYPe.

Given the results of experiment 1, 2 and 3, some advises can be made. Experiment 3 shows less positive results than experiment 1 and 2, because the learned parameter setting of experiment 3 isn't optimal. The target scores aren't approximated and the termination criteria aren't met. The learned parameter settings of experiment 1 and 2 do approximate the target scores very closely and meet the termination criteria. Therefore, it is advised to initialize at least a part of the starting population based on a global first parameter setting as discussed in subsection 4.1.1. This guides the algorithm quicker to better match results (see also end of chapter 7).

Using analytical knowledge to initialize the population, experiment 1 and 2 show that the domain can be tuned with a population size that is smaller than advised (see 5.8.1). However, it is sensible to use a larger population size to guarantee the diversity of the population. Besides the population size, experiment 1 and 2 show that the maximum number of generations doesn't have to be related to time (see 5.10.2) when small matching domains must be tuned. The algorithm quickly reaches an optimal solution, so the algorithm can stop after the termination criteria are met. However, for larger domains it can be wise to make the maximum number of generations dependent on the factor time, because the order of the algorithm is not linear but quadratic and the learning time can become to long.

Besides the requirements related to the DataDictionary and conclusions concerning the control parameters, some general requirements were made. Those are discussed next.

- *The automatically learned parameter set must perform at least as well as the parameter set that is tuned manually.*
  The genetic algorithm approaches the target matches almost exactly. This means that the algorithm is able to tune the parameters at least as well as tuning by hand.
- *The learning algorithm must be able to tune the parameters within a reasonable limit of time.*
  Although the communication between Elise and the algorithm wasn't optimal, the genetic algorithm was able to tune the domain within a few hours, instead of about five days. With an efficient communication tool, tuning will even be faster.
- *The amount of training and test cases needed by the algorithm must not be too large.*
  The needed amount of training and test data stays relatively small. However, when the matching domain is large and a lot of properties and parameters must be tuned, it can be difficult for the client to make cases that cover all the properties and parameters. A possible solution is discussed in section 8.1.
- *The chance on success in real life must be high.*
  Considering that the genetic algorithm meets almost all the requirements, the chance on success of this algorithm is high. An additional assumption must be made: the design of the domain, the matrices, gliding scales must be correct and the training and test cases must also be consistent and correct. When those assumptions are met, the following final conclusion can be made:

Machine learning techniques can be of use in tuning the parameters automatically. Automatic tuning with a genetic algorithm will be easier, faster, and cheaper than tuning by hand. Certainly when using analytical learning in addition. In the next section some recommendations are made to use the genetic tuning algorithm for future work.

8.1. Future Work

In the previous section, recommendations are made based on the test results. Besides the recommendations concerning existing functionality, recommendations can be made about the algorithm itself and new functionality of Elise. The complexity of the algorithm is discussed and three new WEIght attributes of Elise are discussed and the consequences of learning those extra parameters are also discussed.

The order of the algorithm depends on the complexity of its different functions. This order is discussed in subsection 5.10.1. In that subsection it is recommended to omit the check on equal strings in one generation, when large matching domains must be learned. However, another option is possible. Therefore, each string must be a real string (instead of an array, the way it is implemented now). Real strings can be kept in a hashtable as hashvalues. Those values must be calculated and will be assigned to each string. To check the occurrence of double strings in one generation, only the hashvalues have to be compared. This check can be done in linear time, which is less complex than near exponential time.

Besides technical recommendations about the algorithm, Elise is expanded regularly. The generalizibility of the genetic algorithm is high, so new parameters can be learned. For example, three new WEIght attributes are developed recently. Those are described below.

NoDemandValue	This WEIght is applied when a demand property does not have a (demanded) value on the offered side.
Maximum WEIght	This WEIght is the maximum achievable WEIght. Specifying this WEIght is useful when a property has to score less than 100%, even when it is a perfect match. To do this, the maximum weight is set to 100% of the points to score and less points are assigned to the match WEIght (or nomatch WEIght). This WEIght only will be used in very specific matching problems.
Normalized WEIght	When the sum of all achieved Weights and all maximum achievable WEIghts are calculated, then those Weights are normalized to this WEIght. The maximum WEIght of this property and all of its sub-properties is replaced by this Weight. This is used to give groups of properties a Weight relative to other groups of properties. For example, education (and all sub-properties) can be given a normalized Weight of 500 and experience a normalized Weight of 1000. This way, they are normalized to this WEIght, independent of the actual WEIghts and the number of (demanded) properties that are filled in.

The genetic algorithm can be used to tune the three new WEIghts besides the current parameters. Of the new WEIghts, the normalized WEIght is the most interesting to learn, because it is different from the other WEIghts, as it is related to a group of properties instead of just one property. However, a disadvantage of learning more WEIghts (or parameters in general) is the number of needed training and test cases. When more combinations of properties can occur, more cases must be offered to the algorithm to cover all those occurrences. It can be a problem for a client to provide the needed amount of cases. When this is the case, real time learning can be used. First, the algorithm will tune the matching application for about 70 percent. This can be done by increasing the maximum allowed errors, so the termination criteria will be met quicker. The algorithm will tune the last 30% real time based on real cases that are valued with target

match scores. Real time the maximum errors will be set tight again.

Real time tuning can also be used when the supply of cases is sufficient. For example, for domains that are subject to continuous fluctuations and a lot of people is busy tuning those domains day in, day out. This costs a lot of effort to maintain such projects, so real time tuning will be an interesting option.

References

Fitzpatric, J.M., Grefenstette, J.J. (1988). Genetic algorithms in noisy environments. *Machine Learning* 3: 101-120.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.* Reading (MA), Addison-Wesley Publishing Company.

Harel, D. (1992). *Algorithmics: The Spirit of Computing.* Reading (MA), Addison-Wesley Publishing Company.

Harik, G., Goldberg, D.E., Cantú-Paz, E., Miller, L. (1999). The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation* 7(3): 231-253.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems.* Ann Arbor (MI), The University of Michigan Press.

De Jong, K.A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* Ann Arbor (MI), University of Michigan (PhD thesis).

Langdon, W.B., Qureshi, A. (1995). *Genetic Programming - Computers using "Natural Selection" to Generate Programs.* London, University College London (research note).

Man, K.F., Tang, K.S., Kwong, S. (1999). *Genetic Algorithms: Concepts and Designs.* Hong Kong, City University of Hong Kong.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms.* Cambridge (MA), MIT-Press.

Mitchell, T.M. (1997). *Machine Learning.* Boston (MA), The McGraw-Hill Companies, Inc.

Quinlan, J.R. (1986). Induction of decision trees. *Machine Learning* 1: 81−106.

Reed, P.M., Minsker, B.S., Goldberg, D.E. (2001). The practitioner's role in competent search and optimization using genetic algorithms. *World Water and Environmental Resources Congress 2001* (conference paper).

WCC Services BV (2001). *Concepts and Components.* Software documentation, Amsterdam.

WCC Services BV (2001). *DataDictionary Guide.* Software documentation, Amsterdam.

Appendices

A. Glossary

Analytical Learning	The input to the learner includes the same hypothesis space H and training examples D as for inductive learning (see below). In addition, the learner is provided an additional input: A *domain theory B* consisting of background knowledge that can be used to explain observed training examples. The desired output of the learner is a hypothesis h from H that is consistent with both the training examples D and the domain theory B. (T.M. Mitchell, 1997)
Backpropagation	The term backpropagation is used to imply a backward pass of error to each internal node within a neural network, which is used to calculate weight gradients for that node. Learning progresses by alternately propagating forward the activations and propagating backward the instantaneous errors.
Building Block	A pattern of genes in a contiguous section of a chromosome, which, if present, confers a high fitness to the individual. According to the building block hypothesis (Holland, 1975), a complete solution can be constructed by crossover joining together in a single individual many building blocks that were originally spread throughout the population. (W.B. Langdon, A. Qureshi, 1995)
Cart/Case Sorting	A knowledge acquisition technique to gain knowledge about relations, orders
Coefficient Genes	Genes that control the activation and deactivation of other (coefficient) genes of the same string.
Complexity	The (computational) effort that is needed for a learner to converge (with high probability) towards a successful hypothesis. (T. Mitchell, 1997)
Computation Time	The time needed by the algorithm to learn an optimal solution.
Control Genes	Genes of which activation is controlled by other (control) genes of the same string.
Control Parameters	Parameters that steer an algorithm.
Convergence	The tendency of members of the population to be the same. May be used to mean either their representation or behavior are identical. Loosely a genetic algorithm solution has been reached. (W.B. Langdon, A. Qureshi, 1995)
Chromosome	see String.

Crossover	Creating a new individual's representation from parts of its parents' representations. (W.B. Langdon, A. Qureshi, 1995)
Decision Tree Learning	Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree. (T. Mitchell, 1997)
Elitist	An elitist genetic algorithm is one that always retains in the population the best individual found so far. (W.B. Langdon, A. Qureshi, 1995)
Encoding	Each string in a population contains information about the solution it represents. This information is encoded. This can be done in four different ways, namely binary encoding, value encoding, tree encoding and permutation encoding.
Expected Value	The expected number of times a string will be selected to reproduce (M. Mitchell, 1996).
Fitness Function	A function that defines the fitness of an individual as a solution for the required problem. In most cases the goal is to find an individual with the maximum (or minimum) fitness. (W.B. Langdon, A. Qureshi, 1995)
Generation	When the children of one population replace their parents in the population, a new generation has arisen. The number of newly created strings is equal to the population size.
Genetic Algorithm	A population containing a number of trial solutions each of which is evaluated (to yield a fitness) and a new generation is created from the better of them. The process is continued through a number of generations with the aim that the population should evolve to contain an acceptable solution. (W.B. Langdon, A. Qureshi, 1995)
HGA chromosome structure	HGA stands for hierarchical genetic algorithm. The representation of the strings of a HGA exists of control and coefficient genes.
Horn Clauses	A clause is any disjunction of literals (is any predicate or its negation applied to any term), where all variables are assumed to be universally quantified. A Horn clause is a clause containing at most one positive literal. (T. Mitchell, 1997)
Hypothesis Space	see Search Space.
Inductive Learning	The learner is given a hypothesis space H from which it must select an output hypothesis, and a set of training examples $D = \{<x_1, f(x_1)>, \ldots <x_n, f(x_n)>\}$ where $f(x_i)$ is the target value for the instance x_i. The desired output of the learner is a hypothesis h from H that is consistent with these training

	examples. (T.M. Mitchell, 1997)
Knowledge Acquisition	This term refers to any technique to gain knowledge to perform specific tasks.
Machine Learning	A computer program/machine is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. (T. Mitchell, 1997)
Matching	In a matching application, demand and supply are compared to each other to calculate the equality percentage (match score) between both sides.
Mutation	Arbitrary change to representation of a string, often at random. (W.B. Langdon, A. Qureshi, 1995)
Occam's Razor	Prefer the simplest hypothesis that fits the data.
Online Performance Analysis	Online performance analysis tracks the current best solution in a single generation without considering previous generations. (P.M. Reed, B.S. Minsker, D.E. Goldberg)
Offline Performance Analysis	Online performance analysis keeps track of the best individual from all generations preceding and including the current generation. (P.M. Reed, B.S. Minsker, D.E. Goldberg)
Parameter Setting	The parameter setting is the setting of for instance the WEIghts, TYPes, MINs, and Maxes of the different match criteria in the PRODUCT.DIC files.
Population	The population is a set of strings.
Reproduction	Production of new members of population from existing members. May be used to mean an exact copy of the original member. (W.B. Langdon, A. Qureshi, 1995)
Search Space	Space, existing of all the possible hypotheses that represent all different (good and bad) solutions of a problem.
Selection	The process of selecting two strings of the population to create offspring.
Simulated Annealing	Search technique where a single trial solution is modified at random. An *energy* is defined which represents how good the solution is. The goal is to find the best solution by minimizing the energy. Changes, which lead to a lower energy, are always accepted; an increase is probabilistically accepted. The probability is given by $\exp(-\Delta E/k_B T)$. Where ΔE is the change in energy. k_B is a constant and T is the *Temperature*. Initially the temperature is high corresponding to a liquid or molten state where large changes are possible and it is progressively

	reduced using a *cooling schedule* so allowing smaller changes until the system *solidifies* at a low energy solution. (W.B. Langdon, A. Qureshi)
String	Normally, in genetic algorithms the bit string, which represent the individual. In nature many species store their genetic information on more than one chromosome. (W.B. Langdon, A. Qureshi, 1995)
Termination Criterion	This is the criterion that must be fulfilled. When it is fulfilled, the algorithm has learned the problem and can stop learning.
Test Cases	The set of cases, which is used to evaluate the learned solution. The set consists of cases with target scores.
Training Cases	The set of cases, which is used to learn a sufficient solution. The set consists of cases with target scores.
Tuning	Adjusting the parameter settings in the PRODUCT.DIC file until the calculated match scores are acceptable using the tuned parameter setting.

B. PRODUCT.DICs

In this appendix the different PRODUCT.DIC files of the temping agency project are given. First, a new global file and the parts that are added to the original PRODUCT.DIC files are given. After that the original resume PRODUCT.DIC is given, then the vacancy PRODUCT.DIC is given three times. The first one is the file before tuning, the second file is tuned by hand and last the file is tuned by the genetic algorithm in experiment 1.

```
# The following must be added to resume/product.dic
# when learning vacancy/product.dic
- tarvac,    DOM(free),  WEI(0, 0, 0, 0),  MIN(1, 0, 1, 0),\
                                            MAX(1000, 0, 1, 0)
      @NL, "Linked vacancy deals"

- tarmatch, DOM(free),  WEI(0, 0, 0, 0),  MIN(1, 0, 1, 0),\
                                            MAX(1000, 0, 1, 0)
      @NL, "Target matches linked to vacancy deals"

- train,    DOM(list),  WEI(0, 0, 0, 0),  MIN(1, 0, 1, 0),\
                                           MAX(1, 0, 1, 0),  VAL(train)
      @NL, "Train or test set"


# The following must be added to vacancy/product.dic
# when learning resume/product.dic
- tarres,    DOM(free),  WEI(0, 0, 0, 0),  MIN(1, 0, 1, 0),\
                                            MAX(1000, 0, 1, 0)
      @NL, "Linked resume deals"

- tarmatch, DOM(free),  WEI(0, 0, 0, 0),  MIN(1, 0, 1, 0),\
                                            MAX(1000, 0, 1, 0)
      @NL, "Target matches linked to resume deals"

- train,    DOM(list),  WEI(0, 0, 0, 0),  MIN(1, 0, 1, 0),\
                                           MAX(1, 0, 1, 0),  VAL(train)
      @NL, "Train or test set"


# The following must be added to global/values as train.nl
1, "TrainingCase"
2, "TestCase"
```

```
# resume/product.dic
- resid,     DOM(num),    WEI(0, 0, NEVER, 1),     MIN(1, 0, 1, 0),\
                                                   MAX(1, 1, 1, 1)
      @NL, "Resume ID"

- emplname, DOM(free),  WEI(0, 0, 0, 0),          MIN(1, 0, 1, 0),\
                                          MAX(1, 0, 1, 0),  SUMmary
      @NL, "Name employee"

- emplpc,    DOM(num),                           MIN(1, 0, 1, 0),\
                                       MAX(1, 0, 1, 0),  VAL(1000, 9999)
      @NL, "Postal code employee"


# Added!
- tarvac,    DOM(free),  WEI(0, 0, 0, 0),        MIN(1, 0, 1, 0),\
                                                   MAX(1000, 0, 1, 0)
      @NL, "Linked vacancy deals"

- tarmatch, DOM(free),  WEI(0, 0, 0, 0),         MIN(1, 0, 1, 0),\
                                                   MAX(1000, 0, 1, 0)
      @NL, "Target matches linked to vacancy deals"

- train,     DOM(list),  WEI(0, 0, 0, 0),        MIN(1, 0, 1, 0),\
                                          MAX(1, 0, 1, 0),  VAL(train)
      @NL, "Train or test set"
```

```
# vacancy/product.dic before learning
# bold properties must be learned
- vacid,          DOM(num),   WEI(0, 0, 0, 0),          MIN(1, 0, 1, 0),\
                                                        MAX(1, 0, 1, 0)
      @NL, "Vacancy ID"

- vacname,        DOM(free),  WEI(0, 0, 0, 0),          MIN(1, 0, 1, 0),\
                                          MAX(1, 0, 1, 0),  SUMmary
      @NL, "Name company"

- poscat,                                               MIN(1, 0, 1, 0),\
                                                        MAX(1, 1, 1, 1)
      @NL, "Position category"

-- pgroup,        DOM(list),  WEI(0, 0, NEVER, 0),      MIN(1, 1, 1, 1),\
                                          MAX(1, 1, 1, 1),  VAL(pgroup)
      @NL, "Professional group"

-- posclus,       DOM(list),  WEI(0, 0, NEVER, 3000), MIN(1, 0, 1, 0),\
                  MAX(1, 1, 1, 1),  VAL(posclus),       MAT(posclus)
      @NL, "Position cluster"

--- position,     DOM(list),  WEI(0, 0, 0, 3000),       MIN(1, 0, 1, 0),\
      MAX(1, 1, 1, 1000),     VAL(position),    TYP(,OR),   SUMmary
      @NL, "Position"

- poslev,         DOM(list),  WEI(0, 0, NEVER, 3000), MIN(1, 1, 1, 1),\
                        MAX(1, 1, 1, 1),  VAL(poslev),      MAT(poslev)
      @NL, "Position level"

- vacpc,          DOM(num),   WEI(0, 0, 0, 0),          MIN(1, 0, 1, 0),\
                                      MAX(1, 0, 1, 0),  VAL(1000, 9999)
      @NL, "Postal code vacancy"

- traveldis,      DOM(num),   WEI(0, 0, NEVER, 3000), MIN(0, 1, 0, 1),\
                                          MAX(1, 1, 1, 1),  VAL(0,9999)
      @NL, "Travel distance in km"

- duration,       DOM(list),  WEI(0, 0, NEVER, 0),      MIN(0, 0, 0, 0),\
                       MAX(1, 1, 1, 7),  VAL(duration),     TYP(,OR)
      @NL, "Duration assignment"

- ppe,            DOM(list),  WEI(0, 0, NEVER, 0),      MIN(1, 0, 1, 0),\
                                          MAX(1, 1, 1, 1),  VAL(yesno)
      @NL, "Prospect on permanent employment"

- hourspw,        DOM(list),  WEI(0, 0, NEVER, 3000), MIN(1, 0, 1, 0),\
                             MAX(1, 1, 1, 1),  VAL(hpw),    MAT(hpw)
      @NL, "Hours per week"

- vacjob,         DOM(list),  WEI(0, 0, NEVER, 0),      MIN(1, 0, 1, 0),\
                                          MAX(1, 1, 1, 1),  VAL(yesno)
      @NL, "Vacation job"

- vacdate,        DOM(num),   WEI(0, 0, NEVER, 0),      MIN(1, 0, 1, 0),\
                                                        MAX(1, 1, 1, 1)
      @NL, "Date vacancy input"
```

```
# vacancy/product.dic after tuning by hand
# bold properties are tuned
- vacid,          DOM(num),   WEI(0, 0, 0, 0),          MIN(1, 0, 1, 0),\
                                                        MAX(1, 0, 1, 0)
      @NL, "Vacancy ID"

- vacname,        DOM(free),  WEI(0, 0, 0, 0),          MIN(1, 0, 1, 0),\
                                          MAX(1, 0, 1, 0),  SUMmary
      @NL, "Name company"

- poscat,                                               MIN(1, 0, 1, 0),\
                                                        MAX(1, 1, 1, 1)
      @NL, "Position category"

-- pgroup,        DOM(list),  WEI(0, 0, NEVER, 0),      MIN(1, 1, 1, 1),\
                                          MAX(1, 1, 1, 1),  VAL(pgroup)
      @NL, "Professional group"

-- posclus,       DOM(list),  WEI(0, 0, NEVER, 4500), MIN(1, 0, 1, 0),\
                  MAX(1, 1, 1, 1),  VAL(posclus),       MAT(posclus)
      @NL, "Position cluster"

--- position,     DOM(list),  WEI(0, 0, 0, 1500),       MIN(1, 0, 1, 0),\
      MAX(1, 1, 1, 1000),     VAL(position),    TYP(,OR),   SUMmary
      @NL, "Position"

- poslev,         DOM(list),  WEI(0, 0, NEVER, 3000), MIN(1, 1, 1, 1),\
                        MAX(1, 1, 1, 1),  VAL(poslev),      MAT(poslev)
      @NL, "Position level"

- vacpc,          DOM(num),   WEI(0, 0, 0, 0),          MIN(1, 0, 1, 0),\
                                      MAX(1, 0, 1, 0),  VAL(1000, 9999)
      @NL, "Postal code vacancy"

- traveldis,      DOM(num),   WEI(0, 0, NEVER, 300),  MIN(0, 1, 0, 1),\
                                          MAX(1, 1, 1, 1),  VAL(0,9999)
      @NL, "Travel distance in km"

- duration,       DOM(list),  WEI(0, 0, NEVER, 0),      MIN(0, 0, 0, 0),\
                       MAX(1, 1, 1, 7),  VAL(duration),     TYP(,OR)
      @NL, "Duration assignment"

- ppe,            DOM(list),  WEI(0, 0, NEVER, 0),      MIN(1, 0, 1, 0),\
                                          MAX(1, 1, 1, 1),  VAL(yesno)
      @NL, "Prospect on permanent employment"

- hourspw,        DOM(list),  WEI(0, 0, NEVER, 1980), MIN(1, 0, 1, 0),\
                              MAX(1, 1, 1, 1),  VAL(hpw),    MAT(hpw)
      @NL, "Hours per week"

- vacjob,         DOM(list),  WEI(0, 0, NEVER, 0),      MIN(1, 0, 1, 0),\
                                          MAX(1, 1, 1, 1),  VAL(yesno)
      @NL, "Vacation job"

- vacdate,        DOM(num),   WEI(0, 0, NEVER, 0),      MIN(1, 0, 1, 0),\
                                                        MAX(1, 1, 1, 1)
      @NL, "Date vacancy input"
```

```
# vacancy/product.dic after learning
# bold properties are learned
- vacid,          DOM(num),   WEI(0, 0, 0, 0),          MIN(1, 0, 1, 0),\
                                                        MAX(1, 0, 1, 0)
      @NL, "Vacancy ID"

- vacname,        DOM(free),  WEI(0, 0, 0, 0),          MIN(1, 0, 1, 0),\
                                          MAX(1, 0, 1, 0),  SUMmary
      @NL, "Name company"

- poscat,                                               MIN(1, 0, 1, 0),\
                                                        MAX(1, 1, 1, 1)
      @NL, "Position category"

-- pgroup,        DOM(list),  WEI(0, 0, NEVER, 0),      MIN(1, 1, 1, 1),\
                                          MAX(1, 1, 1, 1),  VAL(pgroup)
      @NL, "Professional group"

-- posclus,       DOM(list),  WEI(250, NEVER, NEVER, 4350),\
      MIN(1, 0, 1, 0),  MAX(1, 1, 1, 1),  VAL(posclus),     MAT(posclus)
      @NL, "Position cluster"

--- position,     DOM(list),  WEI(450, NEVER, 1050, 3450),\
      MIN(1, 0, 1, 0),  MAX(1, 1, 1, 1000),      VAL(position),\
                                                 TYP(,OR),    SUMmary
      @NL, "Position"

- poslev,         DOM(list),  WEI(-750, -100, NEVER, 4750),\
      MIN(1, 1, 1, 1),  MAX(1, 1, 1, 1),  VAL(poslev),      MAT(poslev)
      @NL, "Position level"

- vacpc,          DOM(num),   WEI(0, 0, 0, 0),          MIN(1, 0, 1, 0),\
                                        MAX(1, 0, 1, 0),  VAL(1000, 9999)
      @NL, "Postal code vacancy"

- traveldis,      DOM(num),   WEI(-1400, 150, NEVER, 350),\
                        MIN(0, 1, 0, 1),  MAX(1, 1, 1, 1),  VAL(0,9999)
      @NL, "Travel distance in km"

- duration,       DOM(list),  WEI(0, 0, NEVER, 0),      MIN(0, 0, 0, 0),\
                        MAX(1, 1, 1, 7),  VAL(duration),    TYP(,OR)
      @NL, "Duration assignment"

- ppe,            DOM(list),  WEI(0, 0, NEVER, 0),      MIN(1, 0, 1, 0),\
                                          MAX(1, 1, 1, 1),  VAL(yesno)
      @NL, "Prospect on permanent employment"

- hourspw,        DOM(list),  WEI(NEVER, NEVER, NEVER, 3000),\
           MIN(1, 0, 1, 0),  MAX(1, 1, 1, 1),  VAL(hpw),    MAT(hpw)
      @NL, "Hours per week"

- vacjob,         DOM(list),  WEI(0, 0, NEVER, 0),      MIN(1, 0, 1, 0),\
                                          MAX(1, 1, 1, 1),  VAL(yesno)
      @NL, "Vacation job"

- vacdate,        DOM(num),   WEI(0, 0, NEVER, 0),      MIN(1, 0, 1, 0),\
                                                        MAX(1, 1, 1, 1)
      @NL, "Date vacancy input"
```

C. Training Cases

Below the used case sets are given. Each case set exists of one resume deal and several vacancy deals. Each first (*italic*) deal will be inserted into the DataDictionary twice, once as a resume deal (numbered with 1099 – 9099) and once as vacancy deal (numbered with 1000 – 9000). The other deals will only be present as vacancy deals. All cases are original (match) cases used by the tuning phase of the temping agency project, except the cases numbered with (a). Those are added (nomatch) cases.

Position Cluster, *Position*, *Position Level*, *Travel Distance in km* and *Hours per Week* are the properties of the PRODUCT.DIC that must be tuned. The *Target Scores* are the obtained match scores using the PRODUCT.DIC file, which is tuned by hand. The *Actual Scores* are the match scores using the PRODUCT.DIC that is tuned by the genetic algorithm during experiment 1.

Case 1

Deal ID	Position Cluster	Position	Position Level	Travel Distance in km	Hours per Week	Target Score	Actual Score
1000	*Administrative employee*	*Administrative employee*	*MBO*	*10*	*33-37*	*100%*	*100%*
1003	Administrative employee	Administrative employee	MBO	12	33-37	98.05%	98.39%
1002	Administrative employee	Administrative employee	HBO	10	33-37	91.22%	90.14%
1004	Administrative employee	Administrative employee	MBO	10	38-42	91.22%	90.57%
1005	Administrative employee	Administrative employee	MBO	12	28-32	89.27%	88.95%
1006	Administrative employee	Medical administrative employee	MBO	10	33-37	86.70%	84.19%
1001	Filing employee	Filing assistant	MBO	10	33-37	78.72%	79.43%

Case 2

Deal ID	Position Cluster	Position	Position Level	Travel Distance in km	Hours per Week	Target Score	Actual Score
2000	*Commercial employee*	*Commercial administrative employee sales*	*HBO*	*25*	*23-27*	*100%*	*100%*
2003	Commercial employee	Commercial administrative employee sales	HBO	28	23-27	98.77%	98.98%
2002	Commercial employee	Commercial administrative employee sales	WO	25	23-27	91.22%	90.14%
2004	Commercial employee	Commercial administrative employee sales	HBO	25	13-17	86.84%	85.85%
2006	Commercial employee	Technical salesman	HBO	25	23-27	86.70%	84.91%
2001	Purchasing agent	Commercial administrative employee purchase	HBO	25	23-27	78.72%	79.43%
2005	Receptionist	Receptionist	HBO	15	18-22	69.95%	70%
2007(a)	Commercial employee	Commercial administrative employee sales	HBO	25	38-42	0%	0%

Case 3

Deal ID	Position Cluster	Position	Position Level	Travel Distance in km	Hours per Week	Target Score	Actual Score
3000	*Manager/executive*	*Chief administration*	*MBO*	*20*	*38-42*	*100%*	*100%*
3002	Manager/executive	Chief administration	LBO	20	38-42	96.01%	95.52%
3006	Manager/executive	Chief administration	HBO	20	38-42	91.22%	90.14%
3004	Manager/executive	Chief administration	MBO	5	43-54	91.22%	90.57%
3003	Manager/executive	Chief administration	MBO	5	28-32	86.84%	85.85%
3007	Manager/executive	Chief day-care center	MBO	20	38-42	86.70%	84.91%
3005	Manager/executive	Chief administration	HBO	25	43-54	80.00%	78.68%
3001	Entrepreneur/manager other	Manager printing office	MBO	20	38-42	78.72%	79.43%
3008 (a)	Accountant	Assistant accountant	MBO	20	38-42	0%	0%

Case 4

Deal ID	Position Cluster	Position	Position Level	Travel Distance in km	Hours per Week	Target Score	Actual Score
4000	*Carer*	*Ward orderly*	*LBO*	*15*	*13-17*	*100%*	*100%*
4002	Carer	Ward orderly	LBO	18	13-17	97.45%	97.89%
4003	Carer	Ward orderly	LBO	17	18-22	89.66%	89.27%
4004	Carer	Ward orderly	primary	15	08-12	87.23%	86.08%
4005	Carer	Maternity care	LBO	15	13-17	86.70%	84.91%
4001	Nurse	Nurse	MBO	15	13-17	69.95%	69.58%
4006 (a)	Carer	Carer individual health services	HBO	15	18-22	0%	0%

Case 5

Deal ID	Position Cluster	Position	Position Level	Travel Distance in km	Hours per Week	Target Score	Actual Score
5000	*Teacher geography*	*Teacher geography (first degree)*	*WO*	*30*	*18-22*	*100%*	*100%*
5005	Teacher geography	Teacher geography (first degree)	WO	33	18-22	98.83%	99.03%
5002	Teacher geography	Teacher geography (first degree)	WO	33	13-17	90.05%	89.6%
5003	Teacher geography	Teacher geography (first degree)	HBO	10	23-27	87.23%	86.08%
5004	Teacher geography	Teacher geography (first degree)	HBO	30	13-17	87.23%	86.08%
5006	Teacher geography	Teacher geography (second degree)	WO	30	18-22	86.70%	84.91%
5001	Coach	Coach communication skills	HBO	30	18-22	74.73%	74.95%

Case 6

Deal ID	Position Cluster	Position	Position Level	Travel Distance in km	Hours per Week	Target Score	Actual Score
6000	*Post sorter*	*Post sorter*	*LBO*	*15*	*38-42*	*100%*	*100%*
6002	Post sorter	Post sorter	LBO	18	38-42	93.46%	93.41%
6003	Post sorter	Post sorter	LBO	17	33-37	89.66%	89.27%
6004	Post sorter	Post sorter	primary	15	43-54	87.23%	86.08%
6001	Packer	Packer machinery (production)	primary	15	38-42	74.73%	74.95%
6005	Assistance worker construction industry	Assistance expert construction industry	MBO	15	38-42	69.95%	69.58%
6006 (a)	Post sorter	Post sorter	LBO	15	18-22	0%	0%

Case 7

Deal ID	Position Cluster	Position	Position Level	Travel Distance in km	Hours per Week	Target Score	Actual Score
7000	*Consultant*	*AA-consultant*	*WO*	*25*	*32*	*100%*	*100%*
7003	Consultant	AA-consultant	HBO	25	32	96.01%	95.52%
7004	Consultant	AA-consultant	WO	30	33-37	89.22%	88.91%
7002	Consultant	AA-consultant	HBO	10	23-27	87.23%	86.08%
7006	Consultant	Agriculture consultant	WO	25	32	86.70%	84.91%
7001	Controller	Controller public transport	WO	28	32	77.49%	78.41%
7005	Staff manager	Staff manager information and communication sector	HBO	25	32	74.73%	74.95%
7007 (a)	Consultant	AA-consultant	MBO	25	32	0%	0%

Case 8

Deal ID	Position Cluster	Position	Position Level	Travel Distance in km	Hours per Week	Target Score	Actual Score
8000	*Process/mechanical operator*	*Process operator*	*MBO*	*18*	*33-37*	*100%*	*100%*
8004	Process/mechanical operator	Process operator	MBO	20	33-37	98.82%	99.02%
8002	Process/mechanical operator	Process operator	HBO	10	33-37	91.22%	90.14%
8003	Process/mechanical operator	Process operator	MBO	20	38-42	90.04%	89.59%
8005	Process/mechanical operator	Machine operator	MBO	18	33-37	86.70%	84.91%
8001	Industrial designer	Industrial designer	LBO	18	23-27	61.57%	60.8%
8006 (a)	Process/mechanical operator	Process operator	MBO	54	33-37	0%	0%
8007 (a)	Editor	Assistant editor	MBO	18	33-37	0%	0%

Case 9

Deal ID	Position Cluster	Position	Position Level	Travel Distance in km	Hours per Week	Target Score	Actual Score
9000	*Administrative employee*	*Administrative employee*	*MBO*	*18*	*33-37*	*100%*	*100%*
9001	Administrative employee	Administrative employee	HBO	18	33-37	91.22%	90.14%
9004	Administrative employee	Administrative employee	MBO	10	28-32	91.22%	90.57%
9002	Administrative employee	Administrative employee bank	MBO	10	33-37	86.70%	84.91%
9003	Filing employee	Filing assistant	MBO	10	33-37	78.72%	79.43%
9006 (a)	Administrative assistant	Office clerk	LBO	10	23-27	61.57%	60.8%
9005 (a)	Administrative employee	Administrative employee	MBO	210	28-32	0%	0%

D. Test Results

In this appendix the full test results of the three experiments are given. First, the average fitness and the maximum fitness is given per generation. Then, the actual match scores of the fittest strings are compared to the target match results. After that, the errors and fitness score of the fittest string are given and finally the learned parameter settings (represented by the fittest strings) are given.

	Experiment 1		Experiment 2		Experiment 3	
Generation	Average fitness	Maximum fitness	Average Fitness	Maximum Fitness	Average Fitness	Maximum Fitness
0	111,448	331,38	133,206	344,814	69,7447	189,647
1	181,35	352,534	135,613	355,591	67,7351	189,647
2	193,936	352,534	191,4	355,681	67,3054	189,647
3	225,785	356,136	238,485	355,994	68,9406	189,647
4	280,398	356,136	283,586	355,994	67,7995	189,647
5	311,952	365,045	331,144	356,017	75,216	189,976
6	321,11	368,379	333,459	356,026	85,2317	189,976
7	319,89	368,379	327,82	356,414	89,2412	227,958
8	321,986	368,379	324,451	356,414	100,02	227,958
9	318,811	380,577	333,705	359,019	117,307	245,536
10	328,873	380,878	323,339	359,019	121,519	257,133
11	331,453	383,496	320,18	359,019	139,275	257,183
12	335,701	383,496	329,511	368,301	160,343	257,183
13	345,644	398,464	329,949	368,639	198,421	268,024
14	339,037	398,464	333,409	371,172	210,154	268,892
15	338,204	398,838	324,17	371,172	220,86	268,892
16	337,598	398,846	330,321	371,172	226,819	276,466
17	342,058	398,846	342,49	374,341	222,801	276,466
18	347,882	398,981	342,63	377,188	223,009	276,466
19	350,459	399,143	341,86	377,34	221,945	276,466
20	347,74	399,143	345,665	377,34	229,129	276,466
21	365,191	399,143	344,707	377,34	237,233	276,466
22	365,53	399,143	349,392	377,509	251,349	279,488
23	361,613	399,209	350,404	377,509	253,149	279,488
24	381,934	399,245	357,848	380,149	259,329	279,499
25	366,029	399,245	361,812	380,567	273,041	279,517
26	394,781	399,245	364,356	383,408	276,386	282,523
27	392,283	399,245	364,133	383,59	278,637	282,532
28	396,826	399,346	374,292	395,581	279,037	282,59
29	396,78	399,348	377,684	398,701	272,159	282,601
30	397,858	399,379	376,816	398,976	280,201	282,601

	Experiment 1		Experiment 2		Experiment 3	
Target Match Score	Actual Match Score	Difference	Actual Match Score	Difference	Actual Match Score	Difference
100	100	0	100	0	100	0
78,72	79,43	0,709999	78,58	0,139999	0	78,72
91,22	90,14	1,08	91,78	0,559998	89,43	1,79
98,05	98,39	0,339996	94,3	3,75	95,94	2,11
91,22	90,57	0,650002	91,26	0,0400009	91,47	0,25
89,27	88,95	0,32	85,57	3,7	87,42	1,85
86,7	84,19	1,78999	84,47	2,23	0	86,7
100	100	0	100	0	100	0
78,72	79,43	0,709999	78,58	0,139999	0	78,72
91,22	90,14	1,08	91,78	0,559998	89,43	1,79
98,77	98,98	0,210007	96,4	2,37	97,43	1,34
86,84	85,85	0,989998	86,89	0,0500031	87,21	0,370003
69,95	70	0,050003	96,84	0,110001	0	69,95
86,7	84,91	1,78999	84,47	2,23	0	86,7
0	0	0	0	0	0	0
100	100	0	100	0	100	0
78,72	79,43	0,709999	78,58	0,139999	0	78,72
96,01	95,52	0,490005	96,26	0,25	95,2	0,810005
86,84	85,85	0,989998	86,89	0,0500031	87,21	0,370003
91,22	90,57	0,650002	91,26	0,0400009	91,47	0,25
80	78,68	1,32	75,89	4,11	75,82	4,18
91,22	90,14	1,08	91,78	0,559998	89,43	1,79
86,7	84,91	1,78999	84,47	2,23	0	86,7
0	0	0	0	0	0	0
100	100	0	100	0	100	0
69,95	69,58	0,369995	70,35	0,400002	0	69,95
97,45	97,89	0,440002	92,54	4,91	94,69	2,75999
89,66	89,27	0,390007	86,71	2,95	88,23	1,43
87,23	86,08	1,15	87,52	0,289993	86,67	0,560005
86,7	84,91	1,78999	84,47	2,23	0	86,7
0	0	0	0	0	0	0
100	100	0	100	0	100	0
74,73	74,95	0,219994	74,84	0,109993	0	74,73
90,05	89,6	0,450005	87,84	2,21001	89,04	1,01
87,23	86,08	1,15	87,52	0,289993	86,67	0,560005
87,23	86,08	1,15	87,52		86,67	0,560005
98,83	99,03	0,199997	96,58	2,25	97,57	1,26
86,7	84,91	1,78999	84,47	2,23	0	86,7
100	100	0	100	0	100	0
74,73	74,95	0,219994	74,84	0,109993	0	74,73
93,46	93,41	0,049995	88,81	4,65	89,89	3,57
89,66	89,27	0,390007	86,71	2,95	88,23	1,43
87,23	86,08	1,15	87,52	0,289993	86,67	0,560005

69,95	69,58	0,369995	70,35	0,400002	0	69,95
0	0	0	0	0	0	0
100	100	0	100	0	100	0
77,49	78,41	0,920006	74,97	2,52	0	77,49
87,23	86,08	1,15	87,52	0,289993	86,67	0,560005
96,01	95,52	0,490005	96,26	0,25	95,2	0,810005
89,22	88,91	0,309998	85,42	3,8	87,32	1,9
74,73	74,95	0,219994	74,84	0,109993	0	74,73
86,7	84,91	1,78999	84,47	2,23	0	86,7
0	0	0	0	0	0	0
100	100	0	100	0	100	0
61,57	60,8	0,77	61,73	0,16	0	61,57
91,22	90,14	1,08	91,78	0,559998	89,43	1,79
90,04	89,59	0,450005	87,81	2,23	89,02	1,02
98,82	99,02	0,199997	96,55	2,27	97,54	1,28
86,7	84,91	1,78999	84,47	2,23	0	86,7
0	0	0	0	0	0	0
0	0	0	0	0	0	0
100	100	0	100	0	100	0
91,22	90,14	1,08	91,78	0,559998	89,43	1,79
86,7	84,91	1,78999	84,47	2,23	0	86,7
78,72	79,43	0,709999	78,58	0,139999	0	78,72
91,22	90,57	0,650002	91,26	0,0400009	91,47	0,25
0	0	0	0	0	0	0
61,57	60,8	0,77	61,73	0,16	0	61,57

	Experiment 1	Experiment 2	Experiment 3
error_Score	0,620588	1,02353	24,7522
error_Range	0	0	30,8824
error_Matches	0	0	30,8824
Error_NoMatches	0	0	0
error_TotalMatches	0	0	30,8824
Fitness Score	399,379	398,976	282,601

```
Original Settings
Position cluster: WEI(0, 0, NEVER, 3000)
Position:   WEI(0, 0, 0, 3000),     TYP(,OR)
Position level:   WEI(0, 0, NEVER, 3000)
Travel distance in km:  WEI(0, 0, NEVER, 3000)
Hours per week:   WEI(0, 0, NEVER, 3000)

Final Settings when Tuned by Hand
Position cluster: WEI(0, 0, NEVER, 4500)
Position:   WEI(0, 0, 0, 1500),     TYP(,OR)
Position level:   WEI(0, 0, NEVER, 3000)
Travel distance in km:  WEI(0, 0, NEVER, 300)
Hours per week:   WEI(0, 0, NEVER, 1980)

Final Settings of Experiment 1
Position cluster: WEI(250, NEVER, NEVER, 4350)
Position:   WEI(450, NEVER, 1050, 3450),  TYP( ,OR)
Position level: WEI(-750, -100, NEVER, 4750)
Travel distance in km:  WEI(-1400, 150, NEVER, 350)
Hours per week:   WEI(NEVER, NEVER, NEVER, 3000)

Final Settings of Experiment 2
Position cluster: WEI(-150, NEVER, NEVER, 4450)
Position:   WEI(-1650, 1000, 750, 3150),  TYP( ,OR)
Position level: WEI(0, 500, NEVER, 3850)
Travel distance in km:  WEI(-750, -300, NEVER, 1200)
Hours per week:   WEI(450, NEVER, NEVER, 2700)

Final Settings of Experiment 3
Position cluster: WEI(NEVER, 450, NEVER, 8400)
Position:   WEI(350, NEVER, NEVER, 1450), TYP( ,INT)
Position level: WEI(NEVER, -250, NEVER, 6950)
Travel distance in km:  WEI(NEVER, NEVER, NEVER, 1200)
Hours per week:   WEI(-450, NEVER, NEVER, 3700)
```

E. Programming-Code Genetic Algorithm

Below the headerfiles of the population and chromosome objects of the genetic algorithm are given.

```
1       // population.h
2       // population object
3
4       class Population {
5               public:
6                       // constructors
7                       Population ();
8                       // construct population object and sets all variables
9                       Population (int rndPopInit, int pVar, int maxErrA, int maxErrR,
10                                      int maxErrM, int maxErrN, float alphaScore,
11                                      float alphaRange, float alphaTotalMatches,
12                                      float pElitism, float pCrossBit, float pMutVal,
13                                      float pMutBit);
14                      // destructor
15                      ~Population ();
16
17                      // public methods
18                      // set the threshold, the number of properties, calculate
19                      // population size and initialize and allocate related variables
20                      // and objects
21                      void setSettings (int nProp, int thres);
22                      // returns the maximum number of generations allowed
23                      int getMaxGen ();
24                      // returns the number of the current generation
25                      int getNGen ();
26                      // reproduce offspring using operators
27                      void reproduce ();
28                      // calculate fitness for every chromosome in chromosome array
29                      void calcFitness ();
30                      // check if termination criteria are met
31                      bool checkTermCrit ();
32                      // log the last population and write results
33                      void logAndWriteLastPop ();
34
35              private:
36                      // private methods
37                      // check if chrom is not yet in newPop
38                      bool notYetInPop (int chrom);
39                      // test if optimal chromosome also meets the termination criteria
40                      // using the testdata
41                      bool testChromosome ();
42                      // replace the actual population by the new population
43                      void replacePop ();
44                      // reset variables
45                      void reset ();
46                      // write actual population to population logPop.txt file
47                      void logPop (bool bTerm);
48                      // write the result to result.txt
49                      void writeResults (int result);
50                      // variate a chromosome based on the initial parameter setting
51                      // (first chromosome)
52                      void variateString ();
53                      // randomly generate a chromosome
54                      void rndGenerateString ();
55
56                      // operators
57                      // select best chromosomes (nElitism of popSize) to copy to
58                      // next generation
59                      void elitism ();
60                      // select two chromosomes based on the roulette wheel principle
61                      void selection();
62                      // determine if the chromosomes must be crossed and cross
63                      void crossover (int chrom1, int chrom2);
64                      // mutate the chromosome
65                      void mutate (int chrom);
```

```
66
67                  // private variables
68                  int nProperties,
69                          nBits,
70                          nValues,
71                          generation,
72                          maxGen,
73                          tempSelection,
74                          tempSteps,
75                          maxMut,
76                          mutSteps,
77                          popSize,
78                          pVariate,
79                          nCross,
80                          nBitMutation,
81                          nValMutation,
82                          nReproduced,
83                          nElitism,
84                          nNewChroms,
85                          bestChrom,
86                          previousBestChrom,
87                          maxErrAllow,
88                          maxErrRange,
89                          maxErrMatches,
90                          maxErrNomatches,
91                          maxFitnessPossible,
92                          threshold;
93                  float alphaScore,
94                          alphaRange,
95                          alphaTotalMatches,
96                          minFitnessAllowed,
97                          pCrossBit,
98                          pMutVal,
99                          pMutBit,
100                         avgFitness,
101                         maxFitness,
102                         minFitness,
103                         sumFitness,
104                         maxFitnessOverall;
105
106                 Chromosome *actualPop;
107                 Chromosome *newPop;
108                 Chromosome *tmpPop;
109
110                 // will become EliseCommunication
111                 Communicate *communicate;
112                 CDictionary dictionary;
113     };
114
115     ////////////////////////////////////////////////////////////////////////////////
116
117     // chromosome.h
118     // chromosome object
119
120     class Chromosome {
121             friend class Population;
122
123             public:
124                     // constructor
125                     Chromosome ();
126                     // destructor
127                     ~Chromosome ();
128
129                     // allocate array with i chromosomes
130                     static Chromosome *New (const int i);
131
132                     // overloaded assignment
133                     // copy parameter setting of &chrom to this chrom
134                     Chromosome const &operator= (Chromosome const &chrom);
135                     // check if parameter setting of &chrom is equal to this chrom
136                     bool operator== (const Chromosome &chrom);
137
```

```
138                 // set interface
139                 // set WEI bits of PropertyNumber kindOfWeight
140                 void setWeightType (int PropertyNumber, EWeightType kindOfWeight,
141                                     EWeightValueType kindOfValue);
142                 // set TYP bits of PropertyNumber mvtype
143                 void setMultipleValueType (int PropertyNumber,
144                                     EMultipleValueType mvtype);
145                 // set WEI value of PropertyNumber kindOfWeight
146                 void setNumericWeight (int PropertyNumber,
147                                     EWeightType kindOfWeight, int theWeight);
148
149                 // get interface
150                 // return WeightValueType
151                 EWeightValueType getWeightType (int PropertyNumber,
152                                     EWeightType kindOfWeight);
153                 // return MultipleValueType
154                 EMultipleValueType getMultipleValueType (int PropertyNumber);
155                 // return NumericWeight
156                 int getNumericWeight (int PropertyNumber,
157                                     EWeightType kindOfWeight);
158
159                 // public variables
160                 int nProperties;
161
162
163         private:
164                 // private methods
165                 // allocate bit and value string with required number of bits
166                 // and values
167                 void initialize (int nProp);
168                 // set the rgtype or mvtype
169                 void encodeType (int bit, int type);
170                 // variate a chromosome based on the initial parameter setting
171                 // (first chromosome)
172                 void variate (int pVar);
173                 // randomly generate a chromosome
174                 void rndGenerate ();
175                 // write the parameter setting of the best chromosome to
176                 // results.txt
177                 void decodeChrom ();
178                 // write the parameter setting of the best chromosome to screen
179                 void decodeChromWriteToProductDic ();
180                 // return parameter of attribute attr of property prop
181                 char * getParameter (int prop, int attr);
182                 // return integer of attribute attr of property prop
183                 int getInteger (int prop, int attr);
184                 // calculate the fitness score of the chromosome
185                 float calcFitness (resultList rList, int threshold, int maxErrAll,
186                                     float aScore, float aRange, float
187                                     aTotalMatches, int nGen, int nStr);
188
189                 // operators
190                 // mutate the bitstring of the chromosome
191                 int mutateBit (float pMutBit);
192                 // mutate the valuestring of the chromosome
193                 int mutateVal (float pMutVal, int maxMut);
194                 // cross the chromosomes
195                 void crossover (Chromosome chrom);
196
197                 // private variables
198                 int nBits,
199                         nValues,
200                         parent1,
201                         parent2;
202                 float expectedValue,
203                         expValPercentage,
204                         fitnessScore,
205                         fitnessPercentage,
206                         errScore,
207                         errRange,
208                         errMatches,
209                         errNomatches;
```

```
210
211                 // bit encoded string and value encoded string
212                 char **bitString;
213                 int *valueString;
214     };
```

F. Pseudo-Code Communication Tool

The genetic algorithm calculates the fitness scores of the strings, based on the matches Elise calculates with the related parameter setting. To let the genetic algorithm communicate with Elise, an efficient communication tool must be implemented (instead of the communication that is used in this internship). Therefore the EliseCommunication object will take care of the communication. This object will contain three functions, which the genetic algorithm can call. The first function, getInitialSettings(), will be called at the beginning of the learning process to initialize population variables. The second function, readInitialString(), will be called after getInitialSettings() to initialize the first chromosome. The third function, match(), is essential for the learning process. This function will be called each time the fitness score of a string must be calculated. This function will execute Elise to calculate the match scores, using the parameter setting of the specified string and will store them together with the related target scores. Below the pseudo-code of those functions is given.

```
1   getInitialSettings ( Population &pop ) {
2   // the GA calls this function to set the number of properties
3   // and the threshold.
4
5     use the Elise API to:
6           get the number of properties of the PRODUCT.DIC;
7           get the threshold;
8
9     use the Population object to:
10          set the number of properties of the PRODUCT.DIC;
11          set the threshold;
12
13    return;
14  }
15
16  readInitialString ( Chromosome &chrom ) {
17  // the GA calls this function to set the initial string of the
18  // first generation.
19
20    use the Elise API to:
21          get the WEIght attributes (NEVER or integer);
22          get the MultiValueType attributes (OR, AND or
23                                                  INTersection);
24          get the related numeric values (when the weight attribute
25                                                  is an integer);
26
27    use the Chromosome object to:
28          set the WEIghts (NEVER or integer);
29          set the MultiValueTypes (OR, AND or INTersection);
30          set the related numeric values (when the weight attribute
31                                                  is an integer);
32
33    return;
34  }
35
```

```
36  match (Chromosome &chrom, bool trainTest ) {
37  // the GA calls this function to let Elise calculate the match
38  // scores with the parameter setting encoded in the string of the
39  // specified Chromosome object. the function returns a list with
40  // pairs of target and actual match scores.
41
42    use the Chromosome object to:
43          decode the WEIght attributes (NEVER or integer);
44          decode the MultiValueType (OR, AND or INTersection);
45          decode the related numeric value (when the weight attribute
46                                                  is an integer);
47
48    use the Elise API to reset the property parameters:
49          WEIghts (NEVER or integer);
50          MultiValueTypes (OR, AND or INTersection);
51          related numeric values (when the weight attribute
52                                                  is an integer);
53
54    for all vacatures for value trainTest:
55          for all related resumes (this means that the target of
56                                   vacancy-resume pair is known):
57                let Elise calculate the match score;
58                store target and actual match result as pair in list;
59
60    return ( list with actual and target results );
61  }
```