# Using Genetic Programming and SimPack to Learn Global Similarity Measures

Diploma Thesis in Computer Science

submitted by
**Manuel Kägi, Winterthur, Switzerland**
**Student number: 99-713-307**

written at the
**Department of Informatics**
**University of Zürich**

**Prof. Abraham Bernstein, Ph.D.**

Supervised by
**Dipl. Inf.-Ing. ETH Christoph Kiefer**

submitted on
**November 8, 2006**

# Abstract

For a growing number of applications good similarity measures are crucial to ensure that the applications works as desired. Similarity measures can be used to find the most similar object to another one, or can be used to perform a categorisation task, whereby the calculated similarity value will be used to determine the category. But manually defining a good similarity measure, especially if complex and domain specific objects have to be compared, can be a difficult task. A lot of domain knowledge combined with knowledge in computer science (namely how these similarity measures work internally) is needed, and there exists no approved methodology to do this. Therefore the global goal in this diploma thesis is, instead of manually defining similarity measures, to learn them and to evaluate the achieved results.

To be able to learn similarity measures, an universal framework is used, the Local/Global Framework. The idea is to use the Local/Global principle to compare complex objects, whereby the local similarity measures and the amalgamation function can be learned. Another precondition for this is to have an evaluation method to estimate a particular similarity measure's soundness. Typically this is done by comparing the similarity measure's results with a so-called gold standard.

To learn, the evolutionary principles observed in nature will be exploited in an artificial evolution. This artificial evolution can be implemented as a genetic algorithm or a genetic programming approach can be used. In the first case parameters of similarity measures will be learned, in the second case, using the genetic programming approach, the algorithms themselves are learned. In both cases the goal is to find similarity measures, which will show only a small deviation to the gold standard. In the case of using a similarity measure to do a categorisation, the goal will be to properly identify the category an object or a pair of objects (the two compared ones) belongs to.

# Zusammenfassung

Für eine wachsende Zahl von Anwendungen ist es essentiell gute Ähnlichkeitsmasse zu haben, um zu erreichen, dass diese Applikationen ihren Zweck erfüllen. Ähnlichkeitsmasse können dazu verwendet werden, das ähnlichste Objekt zu einem gegebenen Objekt zu bestimmen, oder um eine Kategorisierung vorzunehmen, wobei in diesem Fall die berechneten Ähnlichkeitswerte dazu verwendet werden um die Kategorie zu bestimmen. Aber diese Ähnlichkeitsmasse genau zu definieren, vor allem wenn diese komplexe, sachgebietspezifische Objekte vergleichen sollen, kann eine schwierige Aufgabe sein. Gute Kenntnisse über das Sachgebiet, auf das sich die Objekte beziehen, kombiniert mit Informatikkenntissen (über die internen Vorgänge der Ähnlichkeitsmasse) sind nötig und es gibt keine standardisierte Vorgehensweise solche Ähnlichkeitsmasse zu definieren. Deshalb ist es das Ziel dieser Diplomarbeit, anstatt Ähnlichkeitsmasse zu definieren, solche zu lernen und die erreichten Resultate zu evaluieren.

Um Ähnlichkeitsmasse zu lernen, wird ein Framework benutzt, das Local/Global Framework. Die Idee ist es, das Local/Global Prinzip zu nutzen, um komplexe Objekte zu vergleichen. Dabei können die lokalen Ähnlichkeitsmasse und die Vereinigungsfunktion gelernt werden. Eine weitere Voraussetzung ist es, eine Evaluationsmethode zu haben um die erreichte Zweckmässigkeit eines Ähnlichkeitsmasses abzuschätzen. Dies wird üblicherweise gemacht indem die Resultate des Ähnlichkeitsmasses mit einem sogenannten Gold-Standard verglichen werden.

Um diese Masse zu lernen, werden die Prinzipien der natürlichen Evolution ausgenutzt. Diese künstliche Evolution kann als genetischer Algorithmus oder mit dem Ansatz der genetischen Programmierung realisiert werden. Im ersten Fall werden die Parameterwerte für die Ähnlichkeitsmasse erlernt, im zweiten Fall, wo der Ansatz der genetischen Programmierung verwendet wird, wird ein Algorithmus erlernt. In beiden Fällen ist es immer das Ziel, ein Mass zu finden, dass eine möglichst geringe Abweichung zum Gold-Standard aufweist. Wenn die Ähnlichkeitsmasse verwendet werden um eine Kategorisierung durchzuführen, ist das Ziel, die Kategorie, zu der ein Objekt oder Objektpaar (die beiden verglichenen) gehört, korrekt anzugeben.

# Contents

# List of Figures

# List of Tables

# 1. Introduction

In a growing number of applications, good similarity measures are needed to ensure that they run properly and bring a significant benefit for the users. Such applications may be comparison of products from different online shops, case based reasoning applications or data mining tasks.

Unlike for mathematical operations, for similarity measures there exists no exact definition, how big (or how small) the similarity between to objects is. Therefore a lot of different similarity measures exist and a lot of them also can be configured to achieve, that they calculate a similarity value which the user of the application accepts (or expects). For comparison of complex objects a combination of different similarity measures can be used to calculate a similarity value between two objects. In practice this leads to the fact that each similarity measure has a set of parameters which can be set. In case of using a combination of similarity measures the number of parameters increases. Setting all these parameters to a meaningful value is crucial for making successfully use of a similarity measure.

One way is to ask a domain expert (of the domain, the compared objects come from) to tell us how to set the parameters. But this assumes that the domain expert also knows about similarity measures, the meaning of the parameters etc., things not out of his domain, but part of the computer science.

Another method is to learn the parameter values using a machine learning technique. Usually a machine learning technique needs a feedback, which tells whether the learning results are good or bad. In the case of learning similarity measures (or better their parameter settings), this can be achieved by comparing the result of a comparison with a desired similarity value, that the compared objects should have. This desired value could be defined by a domain expert, who (to do this task) needs no knowledge about computer science at all. He just can define that, according to his opinion (using his domain knowledge), two particular objects have a particular similarity value. A whole set of such comparisons, where the desired result is known, is called a gold standard and can be used to give feedback to a learning algorithm.

In this diploma thesis I will show how we can exploit the principals of evolution to learn good similarity measures. Therefore first the genetic algorithm and afterwards also genetic programming

will be used to learn good similarity measures. Also we want to see how we can learn similarity measures that compare complex objects. Therefore the Local-Global principle is used and enhanced to work in a recursive way. This is done because we want to allow that an object's property may be a complex object (that may contain complex attributes again and so on...). To achieve this goal a "Local/Global similarity measure" framework has been developed which is presented in Chapter 2.
Having a similarity measure and a somehow (e.g. by a domain expert) defined gold standard, a method is needed to evaluate how good this similarity measure might be. This is needed to determine the feedback, which the learning algorithm should get. How a similarity measure can be evaluated is described in Chapter 3.
In Chapter 4 the genetic algorithm (GA) will be introduced. The general aspects of the algorithm will be presented, always with a focus to learn similarity measures using a GA. Especially a detailed description how a similarity measures deviation (which should be small) can be transformed into a fitness value (which should be large) will be presented. Chapter 5 shows an evaluation of learning similarity measures using a genetic algorithm.
Afterwards Chapter 6 introduces the method of genetic programming. In Chapter 7 an evaluation of the results, using genetic programming and a genetic algorithm to learn similarity measures, is presented.
Finally a conclusion part and a chapter where some aspects of implementation are presented conclude this diploma thesis.

# 2. The Local/Global Similarity Measure Framework

In this chapter an introduction into similarity measures in general and the Local-Global framework is presented. The Local-Global framework can be used to define such similarity measures and will be needed to learn similarity measures later on. The goal of having such a framework is that one has standardized interfaces to evaluate and configure similarity measures. This will be important when learned parameter settings have to be transformed into a working similarity measure.

## 2.1. Similarity Measures

A similarity measure is a function that calculates a similarity value between two objects x and y.

$$SimValue = Sim(x, y)$$

The calculated value usually has a range from 0 to 1, whereby the meaning of 0 is defined as no similarity at all between the two objects, and the meaning 1 is defined as maximum similarity between the two objects or equality.

### 2.1.1. General Properties of Similarity Measures

For all possible concrete similarity measures one could think about some general properties which they could fulfil or not. These properties could be used to classify similarity measures and help to find a measure that fits for a given domain.

**Reflexivity**

This property is fulfilled when the calculation of the similarity between two equal objects always returns 1.

$$\forall x : Sim(x, x) = 1$$

Strong reflexivity is given when two objects have a similarity value of 1, **if and only if** these two objects are identical.

$$\forall x, y : Sim(x, y) = 1 \rightarrow x = y$$

Usually similarity measures for primitive types, such as numbers or strings, fulfil the strong reflexivity, while similarity measures for complex types including weights may break the strong reflexivity because e.g. one weight could be 0. Furthermore reflexivity is demanded for a similarity measure in most domains.

**Symmetry**

This property describes whether the calculated similarity value depends on the order of the two arguments given to the similarity function. A measure is symmetric if it does not.

$$\forall x, y : Sim(x, y) = Sim(y, x)$$

Symmetry or asymmetry of similarity measures can be demanded. The following example shows a case where asymmetry is demanded. Imagine looking for a storage device with a capacity of 20 GB. A device with a capacity of 40 GB will fulfil your needs. On the other hand if you are looking for a 40 GB device, one which only 20 GB will not be acceptable.

**Monotony**

Assuming that for all possible values that one wants to compare with a similarity measure an ordering relation (<) is defined (e.g. all possible values are numbers). In this case the similarity measure can be monotonic or not. It will be monotonic if (and only if) the similarity between two objects is smaller than all similarities between any of the two and a third object in between them (based on the ordering relation).

$$\forall x, y, z : (x < y < z \vee z < y < x) \rightarrow Sim(x, z) < Sim(x, y)$$

Knowing about this property can help a lot in practice to configure or to learn a similarity measure because it decreases the search space by one dimension.

**Triangle inequality**

Usually the triangle inequality is defined for a distance measure to be a metric. Having three objects and a particular distance between each pair of them, in this case the triangle inequality defines that none of the three distances may be larger than the sum of the two others.

$$\forall x, y, z : dist(x, z) \leq dist(x, y) + dist(y, z)$$

Formulated for similarity measures the triangle inequality would look like this :

$$\forall x, y, z : Sim(x, z) + 1 \geq Sim(x, y) + Sim(y, z)$$

This means that going over a "third edge", instead of calculating the similarity value directly between two objects and adding 1 (the maximum similarity value), may not lead to a larger value. For a reflexive similarity measure the triangle inequality could also be formulated as follows :

$$\forall x, y, z : Sim(x, z) + Sim(z, z) \geq Sim(x, y) + Sim(y, z)$$

## 2.1.2. The Local-Global Principle

If one wants to compare complex objects instead of just strings or numbers this is done mostly using the Local-Global Principle. Assume we have objects with the attributes $a_1$, $a_2$, $a_3$, ... , $a_n$. Defining one single function that uses the attribute values of two objects to calculate a similarity value would be quite complex and not very flexible. So usually one would define a similarity measure for each of the attributes, the local similarity measures [Stahl 2004]. To calculate a similarity between two objects, first the similarities between the attributes have to be calculated. To calculate then the overall similarity, one can just take the average of all attribute results,

$$Sim = \frac{1}{n} \sum_{i=1}^{n} localSim_i$$

or a weighted average.

$$Sim = \sum_{i=1}^{n} \omega_i \cdot localSim_i$$

While $n$ is the number of attributes, $localSim_i$ is the local similarity value of the i-th attributes and $\omega_i$ is the weight for the i-th attribute. Note that $\sum_{i=1}^{n} \omega_i$ must be 1 to ensure that the value of *Sim* stays in the range between 0 and 1.

**Limitations of the weighted average method**

Generally speaking the average or weighted average is the amalgamation function of a (global) similarity measure. But of course these two methods of calculating the global similarity are not the only ones possible, and also with the weighted average method some phenomena are thinkable which could not be represented properly. Thinking of objects representing rectangles, they would each have two numeric attributes length and width. One could define that for his needs, a similar length is more important than a similar width, by setting the weight for the length attribute higher than the one for the width. But if one would want to achieve that two rectangles which are both squares should get a higher similarity he wont be able to do that just by setting useful parameters for the two attributes length and width. Say we have three rectangles with length/width: 4/4, 4/8,

and 8/8. However the setting of the weights is chosen, the rectangle 4/8 will always be at least as similar to the square 4/4 as to the other square 8/8. One possibility to overcome this limitation is to add a third (calculated) attribute ratio, which describes the shape of the rectangle. Another possibility is to create more complex amalgamation functions which would deal with such phenomena.

The first possibility has its drawback that redundant data has to be created before to be able to calculate a proper similarity measure and one must know that he wants to use the aspect of the length/with ration in his measure. This makes this method not applicable for using machine learning techniques to improve the similarity measures.

The second possibility will be treated in the section about genetic programming, where I will use methods of genetic programming to find good amalgamation functions. The idea is, that contrary to using a genetic algorithm to learn weights or any other **predefined** parameters, a genetic program can also evolve an own (not predefined) structure for the amalgamation function.

### 2.1.4. Recursive Similarity Measures

Another limitation of the (traditional) Local-Global principle is that the attributes of an object must have a primitive type to fit the local similarity measure. This is often not the case with more complex data. For comparison of personal data, each person may have an attribute name, surname, date_of_birth and education. Name and surname would be of type string and be non problem for a local similarity measure dealing with strings, but the two other attributes would be complex objects (such as "person" itself). Say date_of_birth consists of the tree attributes year, month and day, while education has as (sub-)attributes education_type and year_of_graduation. One way to deal with

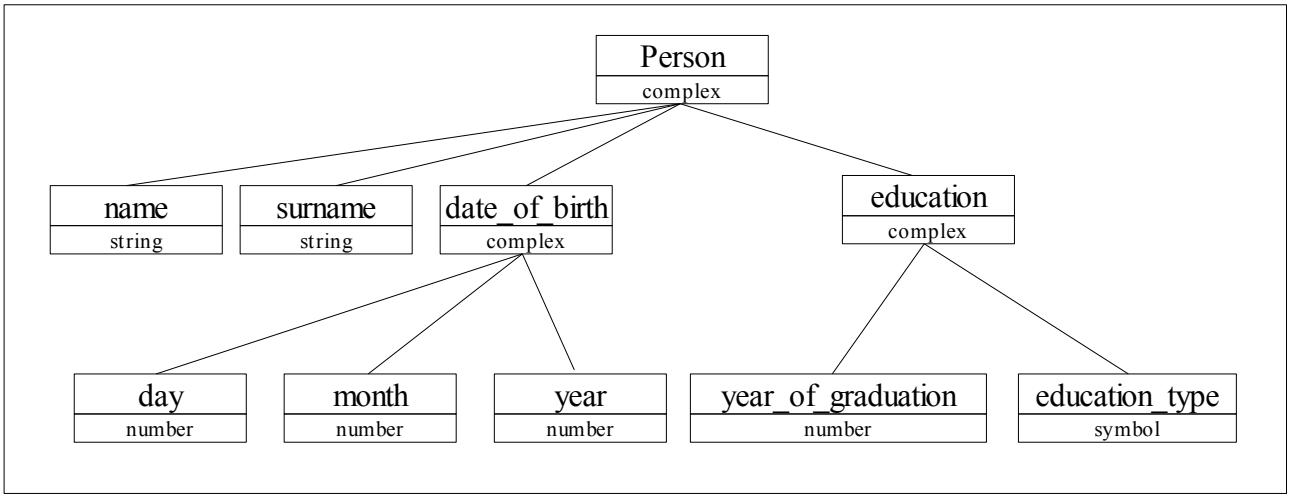


*Figure 1: A possible data structure*

such data is being able to handle a global similarity measure as a local one, so that the local measures for date_of_birth and education (in our example) would be in fact global measures with their own local measures for their attributes.

# 2.2. Types of Local Similarity Measures

Basically the type of a local similarity measure strongly depends on the data type of the values which have to be compared. Basically four different data types can be separated. These are numbers, strings, symbols and composites of these 3 types (i.e. objects). So there are four types of similarity measures, one for each data type.

## 2.2.1. Numeric Similarity Measures

Numeric similarity measures can calculate the similarity of two numeric values. Generally they calculate a distance (in the easiest case the difference) and in a second step transform this distance to a similarity value between 0 and 1.

$$Sim = dist2sim(dist(x, y))$$

Where *dist* is the function calculating the distance between the two values and *dist2sim* is the transformation function. Possibilities for the distance function are taking just the difference or taking the difference of the logarithms of the original values. As transformation function a linear interpolation between two (or more) defined values is possible. Figure 2 shows a transformation with 4 predefined values. Usually the value for *dist=0* is set to 1, and the value for a maximum distance is set to 0. For all distance values bigger than this, the similarity will be set to 0. With distance functions that also return negative values, also asymmetric numeric similarity measures can be configured.

## 2.2.2. String Similarity Measures

String similarity measures can calculate the similarity between two strings. Analogue to the number measures they consist of an internal measure which generates a "raw-similarity"

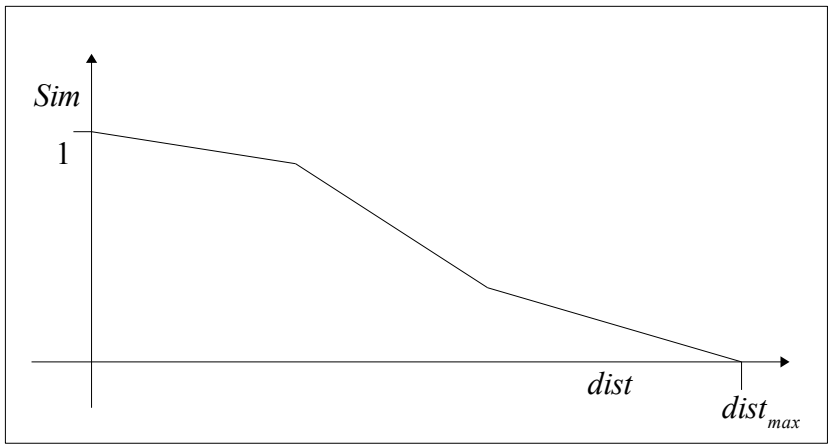


*Figure 2: A possible transformation function with 4 predefined points*

between the two strings, and in a second step this value is transformed to the definitive similarity value with a range 0 to 1.

$$Sim = rawSim2Sim(internalMeasure(x, y))$$

For the internal measure mostly Levensthein is taken, but others are possible of course. For the transformation a linear interpolation between 2 (ore more) defined points can be done (like for the numerical measures). The difference is, that usually the value for *rawSim* = 0 is 0 and the value for *rawSim* = 1 is 1.

### 2.2.3. Symbol Similarity Measures

Symbol similarity measures can be used, when an attribute has a finite number of values that it can adopt. For example the attribute education_type in our persons example could be limited to the values {school, college, university, technical university}. In this case a symbol similarity measure can be used, which returns a similarity value for each combination of values. Internally symbol similarity measures perform no calculation at all (unlike the string- and number similarity) but just lookup a value in a similarity table (see page 10, 2.3.2.3. Similarity Tables). But note, that this similarity table must be defined manually (e.g. by a domain expert) or, in case of learning similarity measures, will be part of the configuration that will be learned.

### 2.2.4. Object Similarity Measures

These measures are always used for complex data types (objects). They consist of one local similarity measure for each attribute of the object (in fact for each attribute that should be taken into account for the similarity calculation), matching to the type of the attribute. If an attribute is again a complex data structure, its similarity measure would be an object similarity measure again. Furthermore, it contains an amalgamation function which calculates one global similarity value from all values calculated by the “attribute”-measures. For example an object similarity measure can compare the date_of_birth of two persons from our example. To do so, it consists of three number similarity measures, one for each attribute which the complex attribute date_of_birth has.

## 2.3. Configuration of Local Similarity Measures

As seen in the previous chapter, every type of similarity measure has its own needs to be configured for a concrete application. So we can say there are (types of) measures on one hand and configurations on the other hand. A type of a measure and a configuration together will result in a concrete similarity measure, ready for being used in an application. Looking forward, the topic of

this theses is learning similarity measures, the configurations also act as link between the learning algorithm and the similarity measures. So actually learned will be configurations (for the types of similarity measures).

## 2.3.1. Which Configurations for which types of Similarity Measures

As we have seen, there are basically four different types of similarity measures and they have all (more or less) different needs of configuration data. Let's see which configuration data can be used for which types of similarity measures.

Note that Distance-to-Similarity Mapping and Similarity-to-Similarity Mapping is quite analogue. The only difference is in the meaning of the input data. While Distance-to-Similarity usually maps a distance of 0 to a similarity of 1 and larger distances will result in lower similarity values, Similarity-to-Similarity maps 0 to 0 and 1 to 1, larger values than 1 will not exist since the input is already a (raw-)similarity. Therefore, a Similarity-to-Similarity mapping is just a special case of a Distance-to-Similarity mapping.

| *Type of measure* | *Possible Configuration* |
|---|---|
| Numeric | Distance-to-Similarity Mapping |
| String | Similarity-to-Similarity Mapping |
| Symbol | Similarity Table or Similarity Tree |
| Object | Object Similarity Configuration |

*Table 1: Types of measures and configurations*

## 2.3.2. Configurations

In this section we will take a look at these configurations in detail.

### 2.3.2.1. Distance-to-Similarity Mapping

A Distance-to-Similarity Mapping consists of an array of similarity values $sim_1$, $sim_2$, $sim_3$, ..., $sim_n$ and an lower- and upper boundary $dist_{min}$ and $dist_{max}$. All input distances lower (or equal) than $dist_{min}$ will be transformed

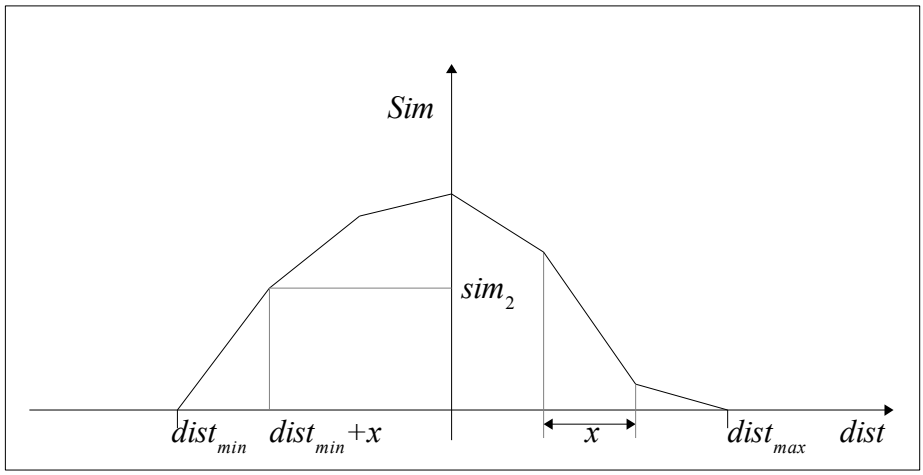


*Figure 3: A general Distance to Similarity mapping, in this mapping negative and positive distances will not be transformed in the same way, a similarity measure would not be symmetric*

to the similarity value $sim_1$. Values bigger (or equal) than $dist_{max}$ will get the similarity value $sim_n$. In Figure 3 (and usually also in practice) $sim_1$ and $sim_n$ is 0. $n$ is chosen to be 7. The width $x$ of one linear section can be calculated as :

$$x = \frac{dist_{max} - dist_{min}}{n-1}$$

The other similarity values of the array will be distributed equally on the remaining interval between $dist_{min}$ and $dist_{max}$. Distances within this interval will be calculated as follows: First, the two nearest (upper and lover) similarity values from the array will be identified and second, a linear (or quadratic...) interpolation between these two similarity values is done to find the value for the given distance.

Note, that using a distance function, which returns also negative values (e.g. a simple difference), this mapping will allow to define asymmetric measures. Also reflexivity is not guaranteed as inherent property of this configuration. There are methods to ensure symmetry, reflexivity and monotony.

Ensuring symmetry can be done by mirroring the similarity values at the *dist*=0 axis.

Ensuring reflexivity can be done by setting the similarity value for *dist*=0 to 1.

Monotony can be achieved by letting the similarity values continuously increase or decrease over the whole range.

#### 2.3.2.2. Similarity-to-Similarity Mapping

As shortly mentioned in the general section, a Similarity-to-Similarity mapping is just a special case of a Distance-to-Similarity mapping. The Similarity-to-Similarity mapping is one, that is ensured to be reflexive (meaning in this case, that the similarity value must be 1 for a "raw-similarity" of 1). Furthermore it is monotonically ascending with an increasing raw-similarity value, and it is ensured that a raw-similarity of 0 will result in a similarity of 0.

#### 2.3.2.3. Similarity Tables

Similarity tables are useful to define similarity values between symbolic attribute values. In the personal data example the attribute education_type can have 4 different values which could be looked at as discrete symbols. So a similarity table for the (symbol-)similarity measure for this attribute is shown in Table 2. A measure using this table would be reflexive because all values in the main diagonal are 1. A measure would not be symmetric because the value in the field School / University is not the same as University / School. A general similarity table for an attribute that's value range

consists out of $n$ different values (symbols) will have the size of $n^2$. If reflexivity is desired, the number of free values decreases to $n(n+1)$ and in case of symmetry to $\frac{n\cdot(n+1)}{2}$. If a measure should get symmetric and reflexive (in fact the most common case) the number of free values is $\frac{n\cdot(n-1)}{2}$. However one drawback for machine-learning of similarity tables is that the search space (in every variant) increases with $O(n^2)$.

| | School | College | University | tech. University |
|---|---|---|---|---|
| School | 1 | 0.6 | 0.2 | 0.2 |
| College | 0.6 | 1 | 0.5 | 0.5 |
| University | 0.3 | 0.5 | 1 | 0.9 |
| tech. University | 0.2 | 0.5 | 0.9 | 1 |

*Table 2: An example of a similarity table, the concrete values might be set by a domain expert or learned by a learning algorithm*

### 2.3.2.4. Similarity Trees

Sometimes the set of symbols, which an attribute can have as its value, can be embedded in a meaningful taxonomy [Gabel 2005]. Thinking of a symbolic attribute vehicle_type in a database of (motor-)vehicles, the value range could be {Limousine, Station-wagon, Cabriolet, Coupé, Van, Open-truck, Closed-truck}. One could arrange these Symbols in a taxonomy shown in Figure 4.

For each node of the tree a similarity value can be defined. So the similarity-value between to types

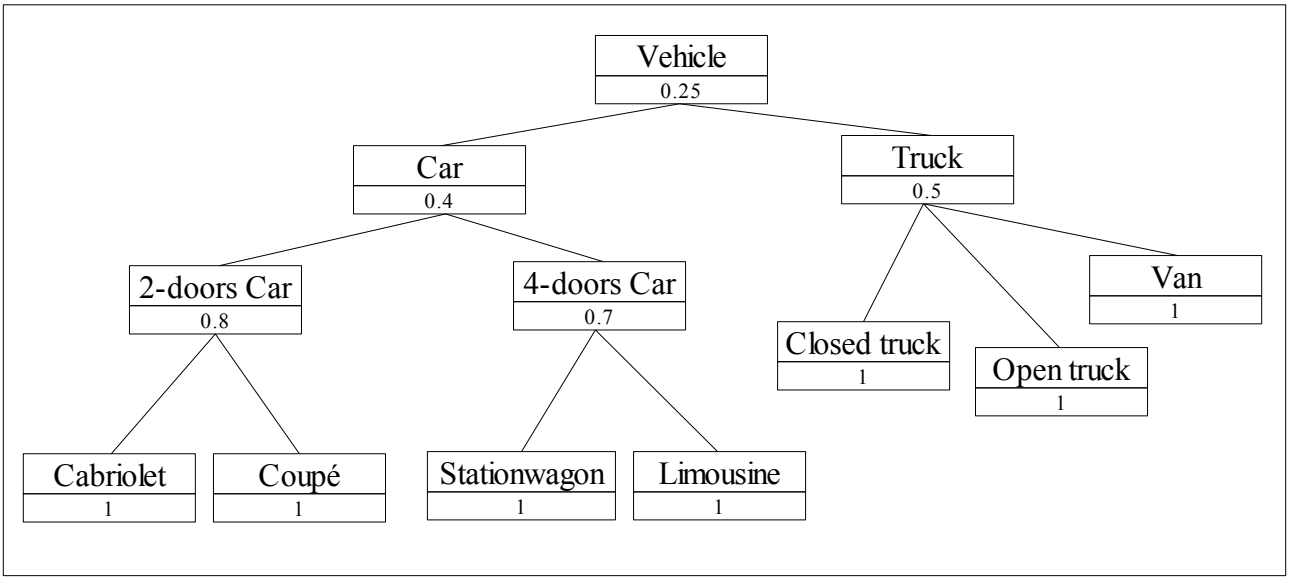


*Figure 4: Similarity Tree for the attribute vehicle_type*

of vehicles can be looked up by finding the Nearest-Common-Parent (NCP) node of the two nodes, which represent the vehicles to be compared. The idea is, that all cars among each other have at least the similarity of car. But if two cars are also both 4-doors cars, they will get the similarity of all 4-door cars, which is usually higher because of the greater specialisation. This strategy, to look

up similarity values between symbols, decreases the amount of parameters to be set to the number of nodes in the tree (unlike the $O(n^2)$ for the similarity tables). Note that an inherent property of the similarity tree is that it will lead to a symmetric similarity measure, because the NCP will be the same for two nodes, independent from the a order of the attributes.

$$\forall x, y : NCP(x, y) = NCP(x, y)$$

If a reflexive measure is desired, the number of parameters shrinks about the number of leaf nodes because their values can be set to 1 in this case. Looking at the example with the vehicle_type attribute, Table 3 shows the numbers of parameters that have to be set.

| | Similarity table | Similarity Tree |
|---|---|---|
| irreflexive | 28 | 12 |
| reflexive | 21 | 5 |

*Table 3: Number of parameters to be set using a symmetric similarity table or a similarity tree*

#### 2.3.2.5. Configuration for Measures for Complex Types

As seen in chapter 2.2.4 an object similarity measure basically consists of a number of (sub-) similarity measures and an amalgamation function. So to configure a concrete object similarity measure, concrete parameters for the amalgamation function and a configuration for each similarity measure, which it contains, is needed. I'll call this an object similarity configuration. If the amalgamation function in the object similarity measure is defined to be a weighted average, its parameter would be an array of the weights. To configure the similarity measures for the object's attributes, an array of configurations can be used (mappings, similarity trees, similarity tables or for a complex attribute another object similarity configuration).

# 3. Evaluating Similarity Measures

Unlike formally well defined algorithms, as for example an equality operator, a similarity measure will never be correct or wrong. It will further be more or less meaningful (or useful) in some cases. In this section we will see how one could evaluate whether a concrete similarity measure is useful or not. Unlike in the last section, the similarity measure itself and the (complex-) objects which it compares, can be considered as black boxes.

## 3.1. Soundness of Similarity Measures

Not only that the results of a similarity measure are generally not mathematically defined and therefore can only be more or less useful, the soundness of a similarity measure can also depend strongly on the users needs. For example, if we have a database of cars (telling us about licence number, colour, shape, manufacturer, engine, year of manufacture, number of seats etc.) and for this a similarity measure which is able to compare two cars, the soundness of one single similarity measure now varies from use case to use case. If the police tries to find a car similar to a witness' description especially the two attributes colour and shape will be important (and maybe licence number in case the witness was able to read parts of it). On the other hand a mechanic looking for a special replacement part, colour, shape and licence number are (usually) unimportant at all, but important would be the manufacturer and maybe the engine. So we can conclude, that a globally useful similarity measure usually does not exist. A similarity measure's soundness will strongly depend on the users needs (and maybe even subjective preferences).

## 3.2. The “Gold-Standard”

One way to estimate the soundness of a similarity measure is to check it against a so called gold standard. This is a set of comparisons with a desired similarity value, defined by the user or a domain expert. In our car-database example this could be a subset of all cars where for each possible pair a desired similarity value is defined.

### 3.2.1. Gold Standard using one data set

In our car database we have just one set of objects and we usually want to compare the members of this set among themselves. So if we have the set of cars $\mathbb{C}$ and the set of car pairs $P = \mathbb{C} \times \mathbb{C}$, for a subset $Q \subset P$ a desired (or say defined) similarity value is given. If the similarity measure for all (or the most) of these pairs returns the same value as defined in the gold standard, when comparing the two cars of the pair, it will be a useful similarity measure (or at least as useful as the values in the gold standard are). Another way to define $Q$ (the set of pairs of which we know the correct similarity values) is to define a subset $G \subset \mathbb{C}$ and $Q = G \times G$. This could also be considered as a quadratic matrix or a similarity table.

### 3.2.2. Gold Standard using two data sets

In another case one might have two databases of car models from two different manufacturers A and B and wants to find the most similar model from manufacturer B to a given model from A (e.g. to buy afterwards the cheaper one). In this case the gold standard should be defined for the pairs $P = A \times B$ while $A$ and $B$ will be subsets of all models in the database from manufacturer A or B, respectively. In this case the matrix usually won't be quadratic. Also it will not be possible to consider the gold standard as a similarity table to check whether it is reflexive. The last section (only one data set) can be seen as a special case of using two sets, namely the one where the two sets are identical.

Another example for this case could be if one wants to find semantically equivalent (but syntactically different) attributes / classes in two different ontologies.

### 3.2.3. Using a "Target function"

A way to create gold standard data is to use a so-called target function. This will be a function (e.g. a concrete similarity measure) that may be defined by a domain expert. Using this function, one could easily generate the gold standard just by calculating the similarity value for each pair that shall become member of the gold standard.

One might think it's senseless to use a target function to create a gold standard to evaluate a similarity measure, because the target function itself would of course be the best fitting similarity measure to this gold standard. So why all that work if also just the target function could be used as the needed similarity measure? There are two cases where this nevertheless makes sense:

1. It could be, that the defined target-function has a bad runtime or a lot of memory usage and the goal will be to find a similarity measure that leads to the same (similar) results but having a better runtime (or memory usage).
2. If (and this is the case in this diploma theses) the goal is just to learn a similarity measure that fits to a somehow created gold standard (and not one that will necessarily be used in practice). This is the case if one is not primarily interested in a practically very useful similarity measure, but only wants to test if the chosen learning mechanism has a good performance (regardless of the fact that the algorithm learns in practice useless or useful measures).

## 3.3. Calculating a similarity measure's deviation from the gold standard

Independent of who has defined the gold standard with which method, a key point of estimating the quality of a similarity measure will always be the calculation of its deviation to the gold standard. This basically consists of two steps: choosing pairs of objects to compare and choosing a meaningful measure for calculating the deviation.

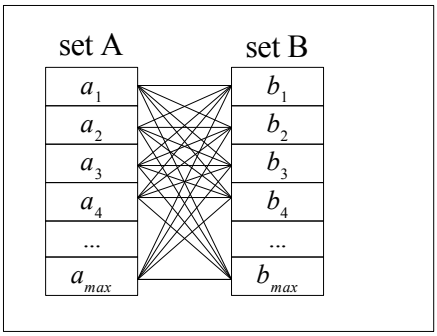

*Figure 5: All possible comparisons*

Choosing the pairs of objects to compare could be done just by using all defined pairs in the gold standard. But often this will lead to a lot of comparisons and so to a long runtime (especially, if the subsets for which the gold standard is defined grow the number of comparisons increases quadratically to the size of the gold standard' s subsets).

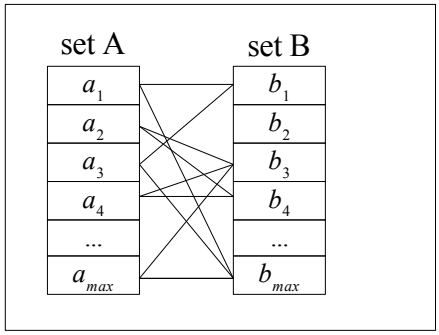

*Figure 6: 2 randomly chosen comparisons for each member of set A*

Another possibility is to choose a value $n$ and then compare each object of subset ***A*** with $n$ objects of subset ***B***. These $n$ objects of subset **B** could be chosen randomly. In the special case that ***A*** is identical with ***B*** it is also possible to choose the next $n$ objects to be compared with the first object. Assuming **A** contains $m$ objects and $a_i$ should be the

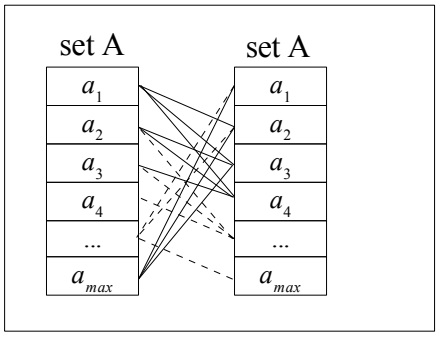

*Figure 7: Choosing the 3 next objects as comparison partners.*

i-th object in the set, $a_1$ would be compared to $a_2$, $a_3$, $a_4$, ... $a_{n+1}$. This will lead to a kind of ring of comparisons and so ensure that each object is exactly $2n$ times part of a comparison.
The details of the measures for the deviation will be described in the next three (sub-) sections.

### 3.3.1. Average Difference

This deviation measure just takes the difference between the value from the gold standard and the value from the similarity measure. It will summarize all absolute values from this differences and divide this sum by the number of comparisons made.

$$avgDiff = \frac{1}{n} \sum_{i=1}^{n} \left| goldStd_i - simValue_i \right|$$

where n is the number of comparisons, $goldStd_i$ is the gold standard value and $simValue_i$ is the calculated value for the i-th comparison.

### 3.3.2. Root-Mean-Square Error

Another possibility is to use the root-mean-square error (rms) as measure for the deviation.

$$rms = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (goldStd_i - simValue_i)^2}$$

Thinking of just two comparisons and both having a difference to the gold standard of 0.1, the average difference and the root-mean-square error both would be 0.1. If the differences would be 0.05 and 0.15 the average difference would still be 0.1 but the rms would grow to 0.1118. This is because the rms “punishes” single bigger differences stronger than it awards single small ones. Due to this fact the value of the rms is always higher than the average difference. This phenomena increases by choosing a bigger exponent in the formula, and shrinks by choosing a smaller one. The two deviation measures could be unified in one formula by introducing another parameter a for the exponent.

$$deviation = \sqrt[a]{\frac{1}{n} \sum_{i=1}^{n} \left| goldStd_i - simValue_i \right|^a}$$

Setting $a$ = 1 will lead to the average difference, $a$ = 2 will result in root-mean-square error, choosing an even bigger $a$ will increase the “punishment” for single large differences.

### 3.3.3. Threshold-Method

The threshold method uses a defined threshold value *t*. All comparisons where the gold standard and the calculated value are both over or both below the threshold are treated as correct. All other comparisons, i.e. if one value is over and one below the threshold are treated as error. All errors will be counted. The amount of errors divided by the total number of comparisons will be the so-called threshold error of the tested similarity measure. This deviation measure can be used if the usage of the similarity measure at the end will be a categorisation task (e.g. finding semantically equivalent attributes in two ontologies).

## 3.4. Using Evaluation to learn Similarity Measures

Note that evaluating a similarity measure, as seen in this chapter, has initially nothing to do with learning a similarity measure. But of course a meaningful evaluation of a single similarity measure is crucial for doing any machine learning. Moreover these methods will also be used for testing the learning results (e.g. against a test set) and can also be used for other tasks different from machine learning. Note also, that the evaluation methods do not yet determine which learning method must be used. In this diploma theses these evaluation methods will be used to learn similarity measures with the genetic algorithm and genetic programming.

# 4. The Genetic Algorithm (GA)

Generally speaking, the genetic algorithm tries to adapt the Darwinian principle of "survival of the fittest" as found in nature to an algorithm. The goal is to get an artificial evolution that generates better and better solutions to a given problem. Unlike the classical engineering approach of development, much less knowledge (theoretically none at all) about the solution's area of expertise should bee needed. Like nature which knows nothing at all about physics, chemistry and so on but nevertheless is able to create very well performing creatures [Dawkins 1986]. As in nature, in artificial evolution a kind a construction plan is encoded to a chromosome. The genetic algorithm then has the task to come up with better and better "construction plans" for the solution one is looking for. The goal is to find a good construction plan and so to find a good solution to a given problem. Additionally it will be presented how a genetic algorithm can be used to learn good similarity measures [Stahl and Gabel 2003].

## 4.1. Terminology

In this section the specific terminology, which is used later on, will be shortly introduced.

- A **chromosome** or **genotype** is the "construction plan" for the solution (as in nature).
- A **gene** is the smallest unit of the **chromosome,** a **gene** has a **gene value**. One or more **gene values** usually represent the value of one specific parameter of the solution.
- A **phenotype** is the solution itself. If the solution is a physical thing the **phenotype** is this thing, if the solution is a algorithm (e.g. a similarity measure) the **phenotype** is this algorithm. The **phenotype** is needed to determine (calculate / meter) if it has the desired properties and so has a high **fitness**. Determining the fitness only by analysing the **genotype** / **chromosome** will not work.
- A **population** is a set of **genotype** or **phenotypes**. Usually before determining the fitness for each **genotype,** its **phenotype** will be built up. **The population** is often used for the whole set of **genotypes** or **phenotypes** a GA works with.
- An **individual** is one **genotype** or **phenotype** of the **population**. The population consists of **individuals**.

- **Fitness** or **fitness value** is a number that describes how good a particular **phenotype** can solve the problem, or how near the **phenotype** is to the ideal solution. The **phenotype's** fitness can be transformed to its **genotype** ("its" means the **genotype** which was used as construction plan for this **phenotype**), or generally the **individual's** fitness.
- The **fitness function** is the function that determines the **fitness** of an **individual**. The fitness function has as argument a **phenotype** or a **genotype**, the first step a fitness function, which gets a **genotype** as its argument, has to do, is to build up the **phenotype**.
- **Selection** is the process that determines which **individuals** may take part at the **reproduction**. The **selection** should use the **fitness** of the single **individuals** to determine that.
- **Reproduction** is the process that builds new **genotypes / chromosomes** out of the **genotypes / chromosomes** of the selected **individuals** (where selected means, that the **selection** process has chosen them to reproduce).
- The **generation** tells how many times already **reproduction** has happened. Unlike nature, in the GA usually all individuals reproduce themselves at one time, so all **individuals** of a **population** will be in the same generation.

## 4.2. Overview of the Genetic Algorithm

If the solution for a problem should be found or improved using a genetic algorithm, it is necessary that it it can be parametrised. If for example an ideal shape for a fuel pipe is desired it could be first parametrised in a way that, for each section of the pipe a parameter defines its curvature. Each parameter will then be analogue to a gene and all parameters together result in a chromosome or genotype of the solution.

Second there must be a method to come from the genotype (i.e. the parameter values) to the working solution (in our example the fuel pipe). This is, generally speaking, the phenotype. Looking at the implementation, the data which the so-called GeneDataInterpreters will get, is the genotype, and the built up similarity measure is the phenotype (in some special cases the genotype and the phenotype may be identical but usually this is not the case).

Third a method to calculate or meter the phenotype's fitness is needed. This is the so called fitness function. The better the fitness value will be, the better will the chances be that this particular phenotype (or its genotype respectively) will be reproduced.

The genetic algorithm then generally repeats following steps in a loop:

1. Create from each genotype of the population its phenotype

2. Determine the fitness of all phenotypes using the fitness function.
3. Select the genotypes which will take part in the reproduction (according to their fitness values)
4. Reproduce the selected genotypes using crossover and/or mutation

Note that the term "population" is important. Using only one genotype will not work because out of one genotype no selection can be done. So the genetic algorithm always works with populations of genotypes.

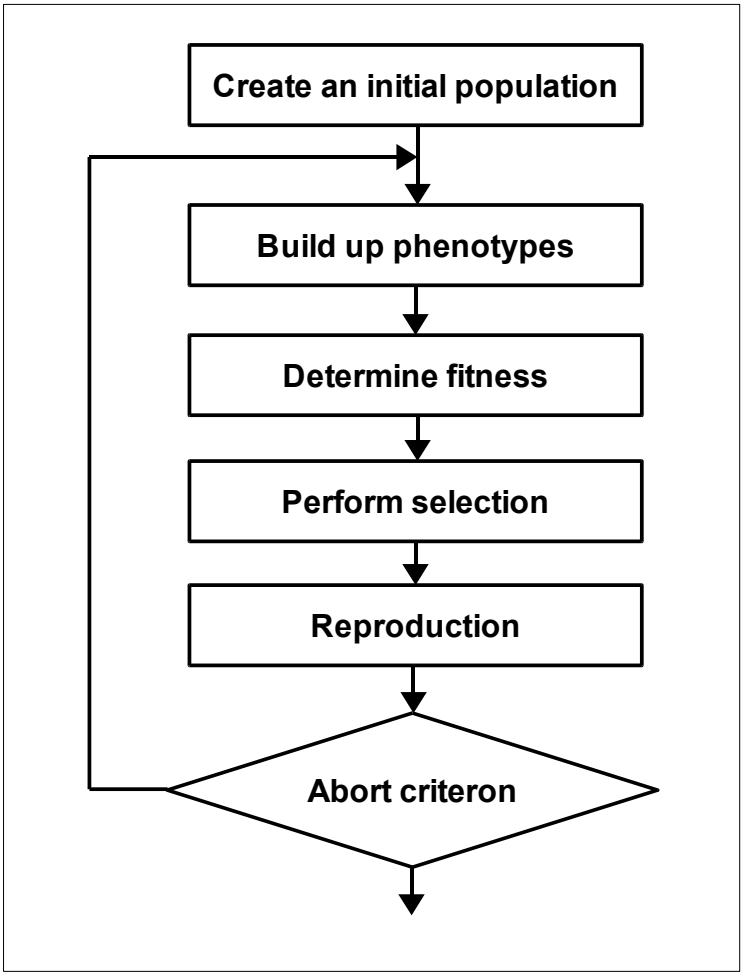


*Figure 8: Flowchart of the genetic algorithm*

## 4.3. The Algorithm in Detail

In this section I will give a detailed description of the major components (see Fig. 8) of the GA and its possible variations. I will not describe the details of the process "Build up phenotype" because this step is very domain specific and in my opinion not part of the GA. But having a method to build up the phenotype is a precondition to use a genetic algorithm. Depending on the concrete implementation, "building up the phenotype" can also be part of the fitness calculation in a way that the fitness function actually has the task to evaluate a genotype. In the case of learning similarity measures using the Local/Global framework (introduced in chapter 2), the so called GeneDataInterpreters will be used to build up the configurations of the similarity measures and than the configurations build up the concrete (working) similarity measures. For more details how the phenotype is built up in this case, see in section "A.2. To learn Similarity Measures" on page 77.

## 4.3.1. Fitness Calculation

Fitness calculation basically consists out of two steps, first one or more domain specific attributes of the phenotype will be metered or calculated and in the second step these values will be transformed to one global fitness value for the phenotype (or its genotype). In the fuel pipe example a property can be the metered flow rate. For evolving similarity measures the central property will be the deviation to a gold standard.

### 4.3.1.1. Calculating the deviation

When the fitness of a similarity measure shall be calculated, calculating its deviation will be the first step. Several methods to to this exist (e.g. calculating the average difference or the root-mean-square error). A detailed description of this task can be found in chapter 3.3.

### 4.3.1.2. Calculating a fitness value from a deviation

One evolving similarity measures, is of course interested in measures which have a small deviation. To achieve this, it is necessary to transform the "raw" deviation value to a fitness value that increases if the deviation decreases. To do this a hyperbolic function can be used.

$$fitness(deviation) = \frac{z}{deviation + a} - b$$

Note the three parameters *a*, *b* and *z,* this common hyperbola comes up with. These can be used to adapt the hyperbola to the concrete needs one might have, transforming a deviation to a fitness. These might be that a defined maximal deviation leads to a fitness value of 0and that a deviation of 0 leads to a defined maximal fitness value (or infinity if *a* is chosen to be 0).

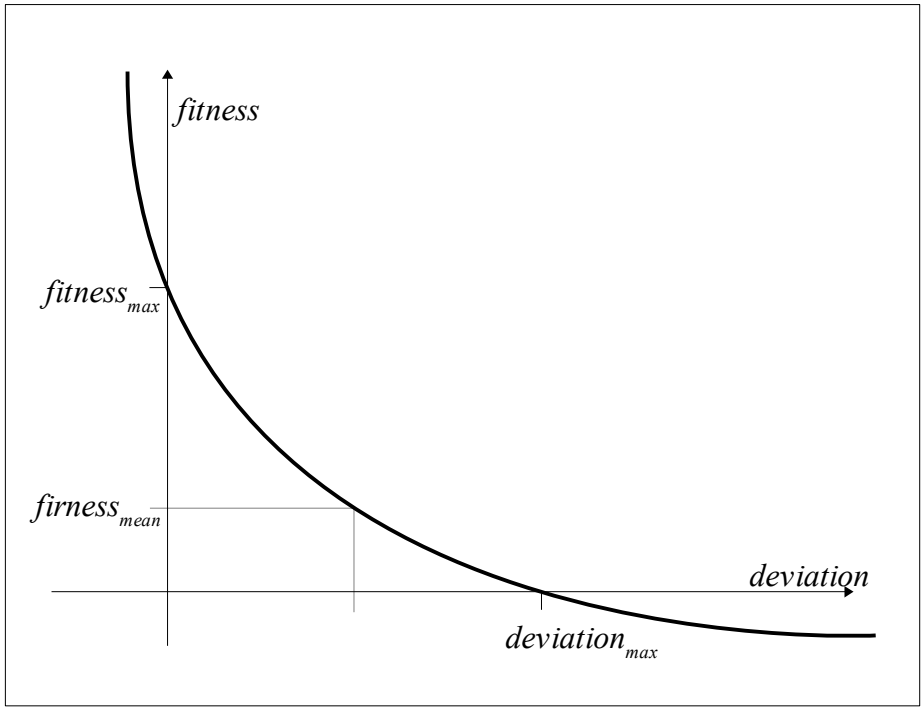


*Figure 9: Hyperbola to transform deviation to fitness*

Since similarity values are always in the range from 0 to 1, the maximal deviation thinkable would be 1 and so one could wish that the fitness of such a measure should become 0 (*fitness*(1) = 0). Another method is to calculate the average distance two randomly chosen values in the interval 0 to 1 will have.

$$avgDist(x) = x^2 - x + \frac{1}{2}$$

This is the average distance two random values will have depending on the value of the first one chosen. Integrating this function from 0 to 1 will lead to the overall average distance.

$$avgDist = \int_{x=0}^{1} x^2 - x + \frac{1}{2} = \frac{1}{3}$$

So it is possible to define $fitness(\frac{1}{3})$ to be 0 because also a random generator will achieve this deviation value .

So if one defines two points which the hyperbola has to cross, namely, *fitness*(0) = $fitness_{max}$ and *fitness*($deviation_{\max}$) = 0, it is possible to set up two equations for the parameters (*a*, *b* and *z*) of the common hyperbola. So a third point of the hyperbola is needed (or from other point of view: can be chosen) to set up the third equation. Having these three points it is possible to calculate the three parameter values. I suppose to define a value $fitness_{mean}$, that defines the fitness function's value for the deviation of $\frac{deviaton_{max}}{2}$ . Choosing this value to be $\frac{fitness_{max}}{2}$ , the resulting function would be a straight line. The smaller this value is chosen the bigger the curvature of the hyperbola will get. The choice of this value will be important for the selection (the next step in the GA), because it actually defines how much fitter a phenotype with a smaller deviation will get. Choosing a small value will increase the evolutionary pressure on individuals which have a smaller deviation than $\frac{deviaton_{max}}{2}$ . Following three equations can be set up:

$$\frac{z}{a} - b = fitness_{max}$$

$$\frac{z}{deviation_{max} + a} - b = 0$$

$$\frac{z}{\frac{deviation_{max}}{2} + a} - b = fitness_{mean}$$

Solving these would show following results:

$$a = \frac{fitness_{mean} \cdot deviation_{max}}{fitness_{max} - 2 \cdot fitness_{mean}}$$

$$b = \frac{fitness_{max} \cdot fitness_{mean}}{fitness_{max} - 2 \cdot fitness_{mean}}$$

$$z = a \cdot (fitness_{max} + b) = \frac{fitness_{mean} \cdot deviation_{max}}{fitness_{max} - 2 \cdot fitness_{mean}} \cdot \left[ fitness_{max} + \frac{fitness_{max} \cdot fitness_{mean}}{fitness_{max} - 2 \cdot fitness_{mean}} \right]$$

Once the three parameters $fitness_{max}$, $deviation_{max}$ and $fitness_{mean}$ have been defined, the three parameters for the hyperbola can be calculated and so, the transformation from deviation to fitness becomes a well defined function which (hopefully) fulfils the users needs.

## 4.3.2. Selection

The main goal of the selection is to give the fitter individuals (i.e. better similarity measures) a better chance to reproduce themselves. To achieve this, usually the roulette wheel strategy is used. A secondary goal of the selection could be to ensure that the fittest (or the fittest 10%) will always reproduce. And as third goal, with a good selection strategy the diversity of the whole population can be raised (or better: a good selection strategy can avoid, that the diversity goes down to 0 and the found individuals stick in a local maxima).

### 4.3.2.1. Roulette-Wheel

This selection strategy can be interpreted as a roulette wheel where each individual has its sector. If the ball falls in a particular individual's sector, this one will be chosen for reproduction. The roulette wheel then is turned as many times as individuals shall be selected for reproduction. Unlike a roulette wheel in the casino, the size of each sector is proportional to the individual's fitness, that the sector represents. In that way the fitter individuals will have a bigger chance to be selected. As variation of this strategy

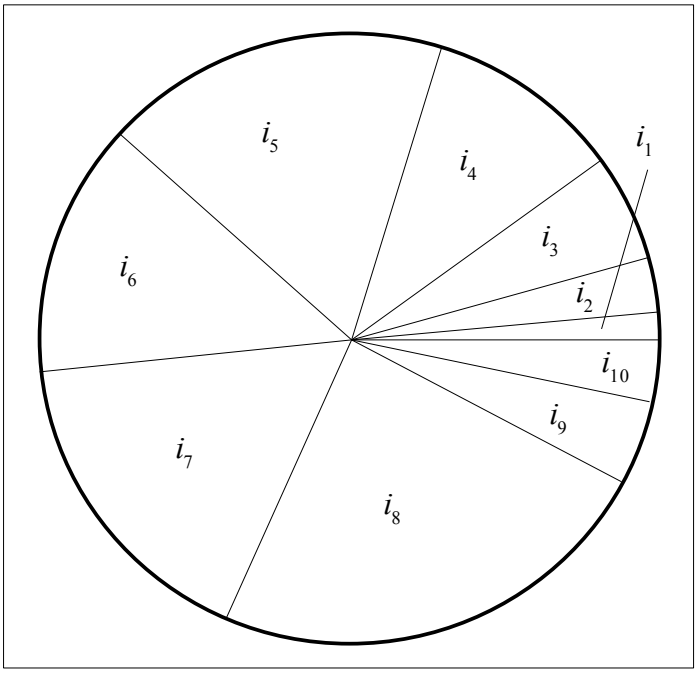


*Figure 10: A possible roulette wheel for 10 individuals*

one could enforce that a particular individual can only be selected once. Using this strategy, it will for none of the individuals ( not even for the fittest ) be guaranteed, that it gets selected.

#### 4.3.2.2. Elitism

This selection strategy just defines that the best $n$ or the best $n$% individuals (the elite) of the whole population will be selected. Individuals which are not part of the elite will have no chance to get selected. As benefit this strategy guarantees that the best performing (i.e. the fittest) will never "get lost". A big drawback is, that in most cases, using this selection strategy will lead to a fast loss of diversity in the population. The diversity of the population is important to avoid that the whole population will become stuck in a local maxima. Another point is that this strategy is far away from the evolutionary processes in nature, which (as mentioned at the begin of this chapter) are the intention of the genetic algorithm. So an elitism strategy will hardly be used as single selection strategy, but could bring benefits when used in combination with other strategies.

#### 4.3.2.3. Equal Chance for all

Like elitism this strategy is quite simple and hardly usable as single selection strategy (even less than elitism). It just ignores the fitness of the individuals and selects randomly some individuals to reproduce. Due to this, using only this strategy will result in a totally random evolution not guided by the fitness function. On the other hand the problem that he population's diversity shrinks will not occur with this selection strategy.

#### 4.3.2.4. Combination of the Strategies

The different benefits of the different selection strategies can be exploited by using a combined strategy. Imagine a box having a capacity of $c$ where the selected individuals will be put into. Now this box could be segmented into three (or more) parts having each a defined capacity $c_1$, $c_2$ and $c_3$ in a way that $c_1 + c_2 + c_3 = c$. Having this done each selection strategy is allowed to fill up its box. Note that the population is not segmented, all selection strategies use the same population as pool of individuals. Think of a population of 100 individuals. For example one might think that he wants to guarantee the selection for the Top-10 performers. Second he wants to use the "natural" strategy of the roulette wheel. Third he wants to have 20 places left for randomly chosen individuals. Doing so can help to overcome the Exploration vs. Exploitation trade-off. The elitism strategy enforces exploitation (of well performing genetic material) and the "equal chance" strategy pushes the

exploration of the whole search space, because also individuals "travelling" through a region (of the search space) where fitness is low, have a chance to survive and reproduce.

### 4.3.3. Genetic Operators

The task of a genetic operator is to perform the reproduction of the selected individuals. Normally the amount of selected individuals is smaller that the amount of individuals in the whole population. In this case the genetic operator has also to ensure that the size of the population increases again up to the original population size. Otherwise the population would shrink in each cycle of the GA and so the reproduction would not be sustainable.

Another method to ensure a lasting population size is to allow that some individuals may be selected several times in the selection phase. So the selection can produce as many selected individuals as the population size is. This brings the following benefit: the genetic operator has not to deal with the question which individual can reproduce how many times. The selection strategy is more appropriate to decide this because it is anyway dealing with the fitness values. So the selection strategy can not only decide whether an individual may take part of the reproduction or not, moreover it can also decide which individual may reproduce how many times (fitter ones may have more children). On the other hand the genetic algorithm must not care about fitness values at all. The two most common genetic operators are crossover and mutation [Pfeiffer and Scheier 1999]. Basically they adapt two methods of reproduction that also can be observed in nature: sexual reproduction and asexual reproduction. Both can be implemented in a genetic algorithm. Sexual reproduction is done by crossover, asexual reproduction by mutation. These two operators will be explained in the following two sections.

#### 4.3.3.1. Crossover

For this genetic operator two individuals A and B are needed and their genes will be combined. In crossover this is done by generating randomly a crossover point. This will be just a point in the chromosomes of the

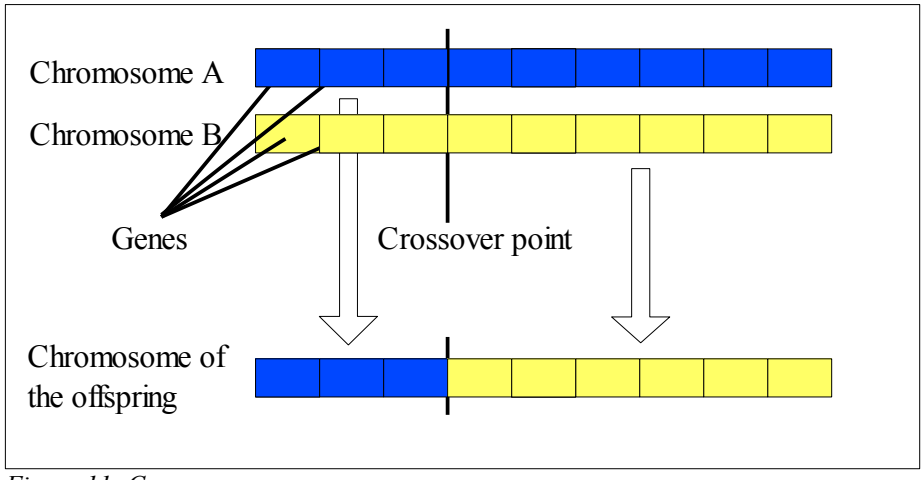


*Figure 11: Crossover*

partners. To create the chromosome of the offspring all genes before the crossover point are taken from chromosome A and all genes after the crossover point are taken from chromosome B. A parameter could define whether partner B must also be part of the selection or if for the partner B any individual from the whole population is allowed. One of the two partners at least must be part of the selection (otherwise the selection won't make any sense).

#### 4.3.3.2. Mutation

Another method of reproduction is just to mutate one or more genes of a chromosome and letting the so created chromosome be part of the offspring generation. If a single gene would be a bit, the mutation of one gene would just be flipping it. There are two ways to select the gene(s) for mutation. First one or more genes on the chromosome are randomly chosen and then exactly these genes mutate. Mostly only one point is chosen and so exactly one gene mutates. Another way is to define a chance of mutation for every gene (e.g. one divided by the length of the chromosome) and then letting each gene having this chance to mutate. For me this way seems to be more like nature works because the genes may mutate independently from each other.
If the value of the genes are not just binary but maybe double values with a range from 0 to 1, the mutation of one gene can be done by increasing or decreasing its value about a certain (small) step. This step again can be generated randomly and may have a limit. This way, when working with double-valued-genes, its possible to define a maximum mutation impact.

#### 4.3.3.3. Combination of the genetic operators

The above introduced two operators can also be used both together. They can be combined in a way that sometimes mutations and some other times crossover is performed. But it is also possible to let the two operators work independently from each other. To achieve this one could define an overall chance that crossover happens and a chance that mutation happens. If an individual then is chosen not to be reproduced by crossover it first will be just copied from the original one. Afterwards it will maybe perform mutation with the same chance of mutation all individuals have.

## 4.4. Parameters for the genetic algorithm

In this section a summary about all the options and variations of the GA, seen in this chapter will be presented. Moreover we will see a list of all possible parameters, their meaning and value ranges. Also some evaluations will be shown, which have been made to find healthy values for the different parameters.

### 4.4.1. List of parameters

**General parameters**

*populationSize*, the population's size, usually in the range of 50 to 1'000).

- *generations*, defines how many generations will be generated, usually in the range of 100 to 1'000).

**Parameters for the fitness function**

- *testsPerRecord*, defines how many comparisons per record are done to estimate the deviation (1 – number of records).
- *testPairCreation*, defines how the comparison partners are selected (the next *n* records, randomly once or randomly new each generation).
- *deviationType*, type of deviation calculated (average difference, root-mean-square error or threshold).
- *tresholdValue,* value of the threshold (if the threshold method is used; range 0 – 1.
- $deviation_{max}$, deviations greater or equal than this parameter value will get fitness 0; range 0 – 1.
- $fitness_{max}$, the fitness value an individual with deviation 0 gets (e.g. 100).
- $fitness_{mean}$, the fitness value an individual gets that has the half deviation of the defined maximum deviation (e.g. 1, must be smaller than the half of the maximum fitness).

**Parameters for the Selection**

- *eliteSize*, size of the elite in percent of population (members of the elite will have a guarantee to be selected), 0% – 100%.
- *grr*, guaranteed reproduction rate for the elite (member of the elite will be a least as many times selected, 1 or more (if the elite e.g. is defined as the top 10%, the guaranteed reproduction rate can be 10 as maximum, then the whole "box of selection" is full).
- *jokers*, how many individuals will be selected just randomly (Equal chance for all) in percent of the population size.

Looking at the model with the boxes of selected individuals (introduced in the section 4.3.2.4. Combination of the Strategies) the box for elitism will have the capacity of $eilteSize \cdot grr$ percent, the box for "Equal chance for all" will have the capacity of *jokers* percent. The remaining capacity will be used for roulette wheel selection. Note that $eliteSize \cdot grr + jokers$ must be smaller (or equal) than 100%.

**Parameters for the reproduction**

- *crossoverRate*, the chance of each individual to get reproduced by a crossover operation with another randomly chosen individual.
- *mutationRate*, the chance that a (somehow) reproduced individual to mutate.
- *maxMutationImpact*, the maximum impact a mutation can have (assuming the genes represent double values).

## 4.4.2. Parameter values

In the last section of this chapter we will see the concrete values, I used for the parameters. The value for *deviatonType* depends on the evolution's goal. If the goal is to find a similarity measure with a small deviation, "root-mean-square error" or "average difference" is used. If the goal is to use the found similarity measure for a categorisation task (e.g. alignment finding) the threshold method is used. Depending on the deviation type, $deviation_{max}$ can be defined. It will be $\frac{1}{3}$ if the first case (as seen before), and calculated in the case of a categorisation task. (In this case the $deviation_{max}$ will be set to the threshold error, which a measure would achieve by always returning 0). Second an present an experiment will be presented, that has been done to find good parameter values. The experiment, that will be shown, has been done to find good values for *maxMutationImpact*. This parameter is chosen because it is not part of the "standard" GA parameter set.

| Parameter | Value |
|---|---|
| *populationSize* | 50...250 |
| *generations* | 50...500 |
| *testsPerRecord* | 1...50 |
| *testPairCreation* | random |
| *thresholdValue* | 0.5 |
| *deviatonType* | [domain dep.] |
| $deviaton_{max}$ | [calculated] |
| $fitness_{max}$ | 100 |
| $fitness_{mean}$ | 1 |
| *eliteSize* | 5.00% |
| *grr* | 2 |
| *jokers* | 50.00% |
| *crossoverRate* | 0.5 |
| *mutationRate* | 0.5 |
| *maxMutationImpact* | 0.2 |

*Table 4: The parameter values for the GA*

Figure 12 shows the achieved deviation depending on the maximum mutation impact. One can see that the maximum mutation impact should not be set too small. It shows also nicely that all values above 0.1 have lead to a quite good performance of the GA. This actually is one of the benefits working with genetic algorithms: they are quite stable against the settings of the parameters. In practice of course this makes the work a lot easier. Doing such analysis for other parameters brings

very similar results. Some extreme values will decrease the GA's performance but there is always a quite large range of healthy values.

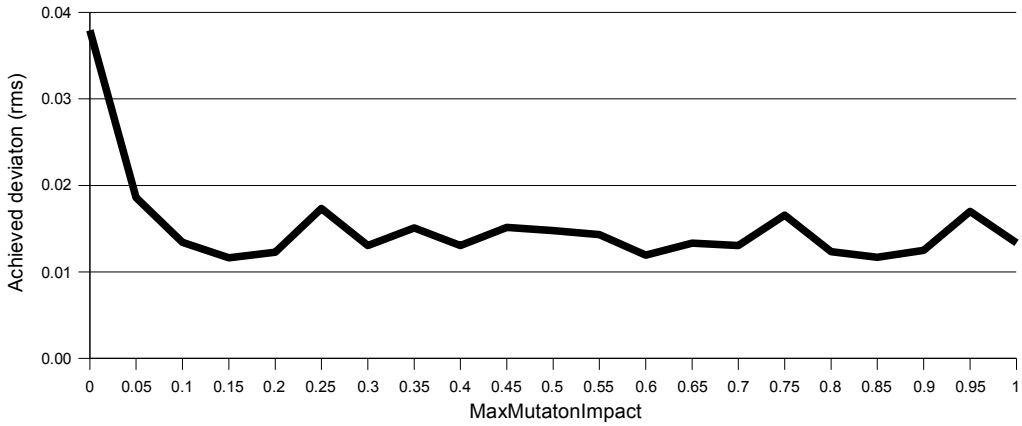


*Figure 12: The achieved deviation after 200 generations with a population of 50*

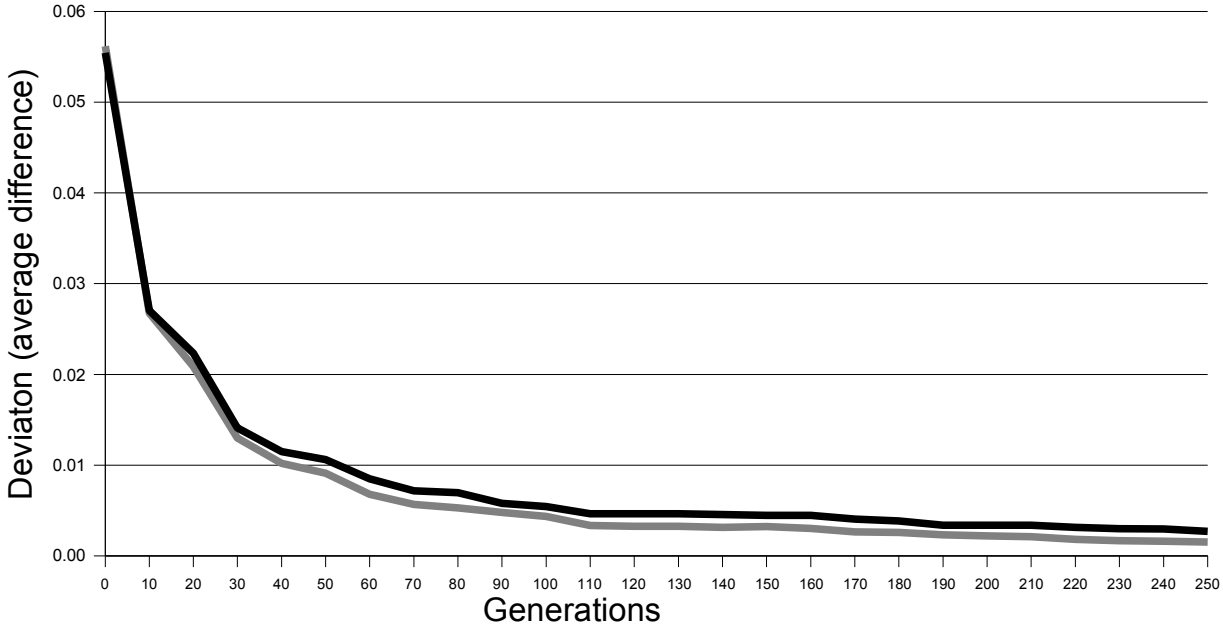


*Figure 13: Achieved deviation increasing the number of generations (learning rate)*

Looking at Figure 13 one can see the impact of the parameter *generations*. On this diagram one can see that the learning steps at the beginning are big and then the improvements get smaller and smaller. This is a typical learning rate. And of course (unlike the things mentioned to maxMutationImpact) this parameter has a significant influence on the GA's performance. The grey line shows the deviation from the training set, the black line shows the deviation from the so-called test set. The test set is another gold standard, but hidden to the GA. This is done to verify if the

learned measure is also meaningful for data which was not taken into account during the learning phase.

# 5. Learning Performance with the Genetic Algorithm

Putting all together, the Local-Global framework from Chapter 2, the methods to evaluate a similarity measure (Chapter 3) and the genetic algorithm from chapter 4, one is able to learn similarity measures using a given gold standard. In this chapter I will present some evaluations of the learning performance and the achieved deviations. In the first part I will show some experiments using a database of (shop-)products. In this part also the principle of training- and test set and the cross validation will be introduced.

In the second part of this chapter some experiments, that have been done with ontologies, will be presented. The task was to find a correct alignment from properties of one ontology to the properties of another ontology.

## 5.1. Using a Product Database

The article data set consists out of 2850 objects (i.e. the products). Each product firstly has 6 properties (actually they have more but in these experiments only 6 are used). Figure 14 shows the structure of a product-object. Firstly it is flat, later on a tree-like structure will be used to evaluate the recursive similarity measures (see section 2.1.4. Recursive Similarity Measures). The symbolic attribute “type” defines what kind of product (food, fertilizer etc.) it is. The attribute “group” defines for which kind of animal a product is (the product set is from a pet shop). The two numeric attributes “min” and “max” define the minimum and maximum stock of inventory of the product.

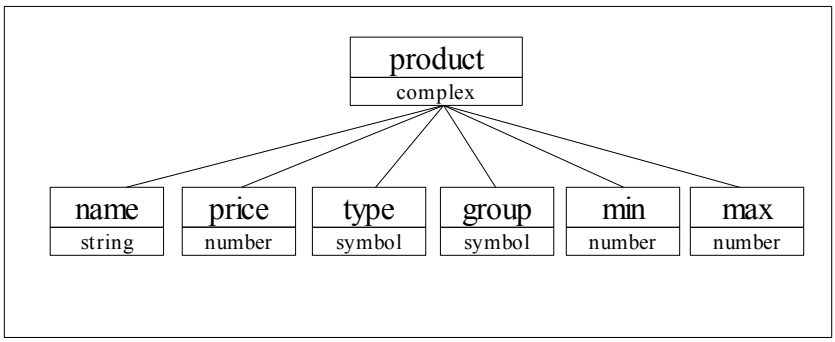


*Figure 14: The product's attributes*

### 5.1.1. Validation using a Test-set

Assume having a gold standard, that tells the desired similarities among e.g. 100 products, one problem that can occur when learning with a genetic algorithm, is that a similarity measure could be learned that is very specialised to this gold standard but will not perform well for the rest of the products. This is the so called over-fitting problem. To minimize this problem, the gold standard can be defined for more products, but this may be a hard work, and a simply larger gold standard would increase the runtime. So it is necessary to estimate how big (or hopefully how small) the over-fitting effect is. Therefore, the learned similarity measure is checked against another gold standard, the so-called test set. This set was not used for the learning algorithm itself, and so a good performance on the test set would hardly be a result of over-fitting. Analysing the training- and test set performance also helps to initially find a healthy number of generations. Figure 16 shows the root-means-square error of a similarity measure against the training- and test set. While after 200 generations the training set error still decreases, the test set error increases again. So after this point unhealthy specialisation starts. This experiment has been done with a relatively small population of only 25 individuals. Figure 17 shows the same experiment, using a larger population

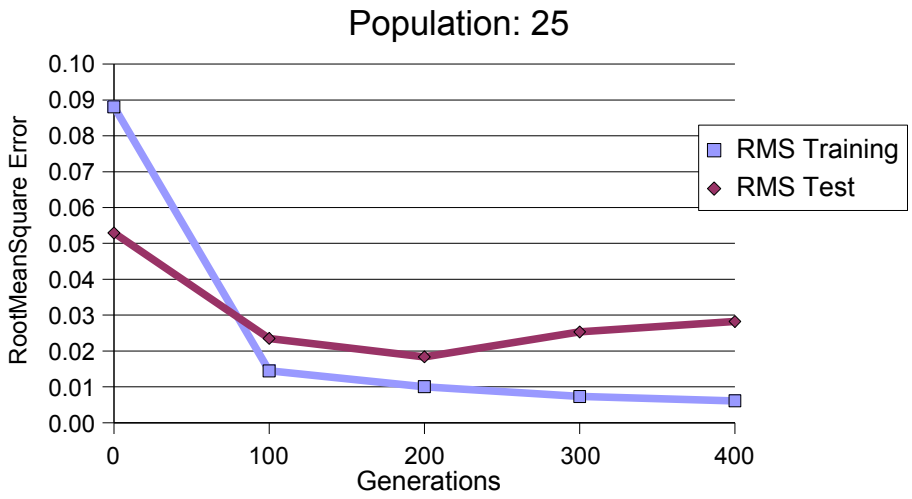


*Figure 16: Training- and Test set performance*

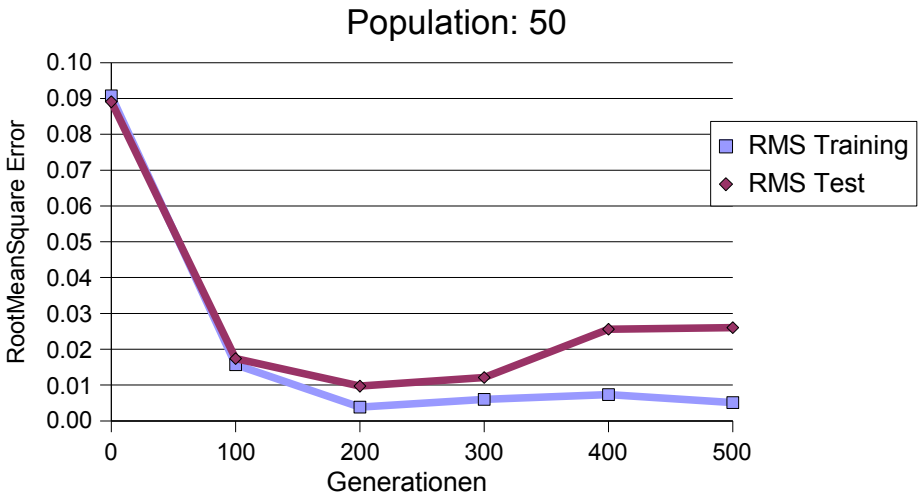


*Figure 17: Training- and Test set performance*

of 50 individuals. One can see that the effect is a bit weaker and seems to start later. For these experiments a quite small training set has been used (and so a small gold standard has been used to learn). In the following experiments, where a cross-validation is done, a larger training set will be used to even more decrease the specialisation effects.

## 5.1.2. Cross-Validation

Another method to check the soundness of the learning method and the learned similarity measures is to perform a so-called cross validation. This method uses the test set evaluation in a special way and generally has two different variants, leave-one-out and k-fold cross validation.

### 5.1.2.1. Leave-One-out Cross Validation

The leave-one-out method works as follows: The training set contains all objects except for one single object. This object (the "left-out" one) then is the test set. Afterwards every object of the whole set will be once this special object, so the learning-cycle runs as many times as objects are contained in the whole set. This of course leads (especially for large sets, as our product set) to a very long runtime of the whole cross validation task. Also in my opinion it does not show that the learning algorithm is able to come up with good general measures, having a relatively small training set (since the training set will consist of the whole set except one single object). At last for similarity measures (comparing **two** objects) this will not work. Thinkable is a leave-two-out method, but due to the other drawbacks, all experiments have been done with the k-fold method.

### 5.1.2.2. k-fold Cross Validation

The idea of the k-fold cross validation is to segment the whole set into *k* subsets. Then every segment is once the training set, while all other segments together work as test set. This will lead to *k* runs of the learning cycle. Meaningful values for *k* could be 5 ... 50. Using a very small *k* (e.g. 2) would lead to big training sets and therefore not show the ability of the learning algorithm to work with a relatively small training set. Very large values will increase the runtime because the whole learning cycle will run *k* times. In all following experiments I will use this type of cross validation with a *k* of 10 (10-fold cross validation). Remembering that our product set contains 2850 products, the training set would always contain 285 products and the test set's size will be 2565.

## 5.1.3 Cross Validation Results

All experiments in this section use the parameter settings shown in Table 5. A detailed description of the meaning of each parameter can be found in the last chapter ("4.4. Parameters for the genetic

algorithm" on page 27). The first two experiments shall show the benefits of using "vocabulary knowledge". This is general knowledge one might have about the similarity measure that will be learned, for instance that the local similarity for a particular attribute is symmetric. This vocabulary knowledge can be used to decrease the search space for the genetic algorithm. E.g. for the attribute "price" it can be defined that the similarity value has to decrease monotonic if the price-difference increases. For the symbolic attribute "group" a taxonomy can be defined so that, instead of a (large) similarity table, a smaller similarity tree has to be learned. For "type" it can be defined that the similarity table has to be symmetric and so on. Doing so, we can exploit knowledge about the desired similarity measure, one might already have.

| Parameter | Value |
|---|---|
| *populationSize* | 50 |
| *generations* | 250 |
| *testsPerRecord* | 2 |
| *testPairCreation* | random |
| *thresholdValue* | 0.5 |
| *deviatonType* | RootMeanSquare Error |
| $deviaton_{max}$ | 0.33 |
| $fitness_{max}$ | 100 |
| $fitness_{mean}$ | 1 |
| *eliteSize* | 5% |
| *grr* | 2 |
| *jokers* | 50% |
| *crossoverRate* | 0.5 |

*Table 5: The parameter values used for the for the GA in the next 4 experiments*

Figure 18 shows the results of the 10-fold cross validation without using any vocabulary knowledge. The RootMeanSquare Error (rms) on the y-axis of the diagram shows the achieved deviation to the gold standard, a perfect measure would have a rms of 0. On the x-axis the number of the section that was used as training set is shown (since a 10-fold cross validation divides the whole data set into 10 segments). The two bars show the achieved training set respectively test set error, according to the legend shown in Figure 18. The legend will be valid for all diagrams of results in this

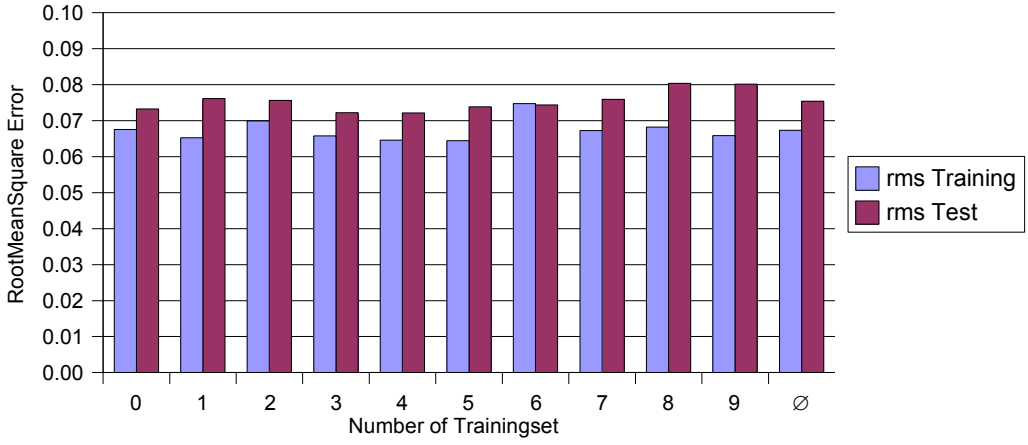


*Figure 18: Cross validation results using no vocabulary knowledge, as deviation the RootMeanSquare Error, that the best learned measure achieved,is shown.*

section (therefore, the legend is left away in the following diagrams). The diagram shows three things:

- That the at the training set learned measures have not too much over-fitting tendencies, the test- and training set errors are for each training set quite the same.
- It shows that with all training sets it is possible to learn a good similarity measure, this depends not too much on the chosen training set.
- The observed deviation is overall not too good , we will see better similarity measures in the experiments with the exploitation of vocabulary knowledge.

For a 10-fold cross validation a 1.6 GHz AMD computer comes up with a runtime of about ¾ hours.

In a next experiment, all the previously mentioned vocabulary knowledge is used and so the search space is smaller. Also the length of the chromosome needed to encode all parameters of the similarity measures is much smaller. Whereby the chromosome contained 63 genes in the last case, it contains only 33 genes when the vocabulary knowledge is exploited. Figure 19 shows the benefits of exploiting this knowledge. The learned measure perform dramatically better after the same learning time. Note that this is actually an optimization task. Without exploiting the vocabulary knowledge, it is also possible to learn as good measures, but it takes much more runtime.

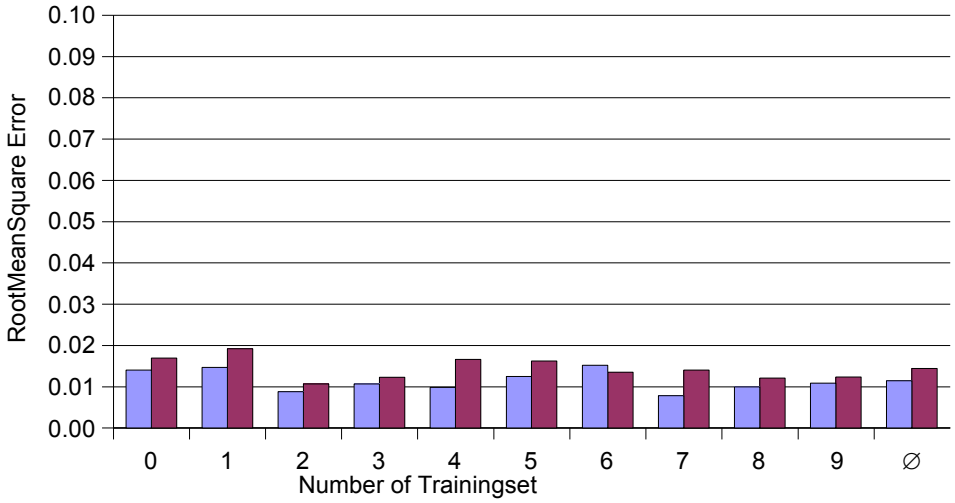


*Figure 19: Results of a 10-fold cross validation using vocabulary knowledge*

**Tree-like data Structure**

In this experiment the performance learning recursive similarity measures should be evaluated. To do so, the data structure of the products to be compared has been changed to get 2 complex attributes. Actually the two symbolic attributes “type” and “group” have been arranged under the

newly created complex attribute "classification" and the two numeric attributes "min" and "max" are merged under the newly created stock-attribute. Looking at the results of the 10-fold cross validation again, it shows, that it works with a very similar performance as the experiment using the flat data structure. Note, that the scale in Figure 21 has changed, compared to the first two experiments, to make the differences between the different runs better visible. Looking at the average performance over all runs, it is slightly worse than the experiment with the flat data structure. For the test set this value was 0.0144 for the flat, and 0.0148 for the recursive case.

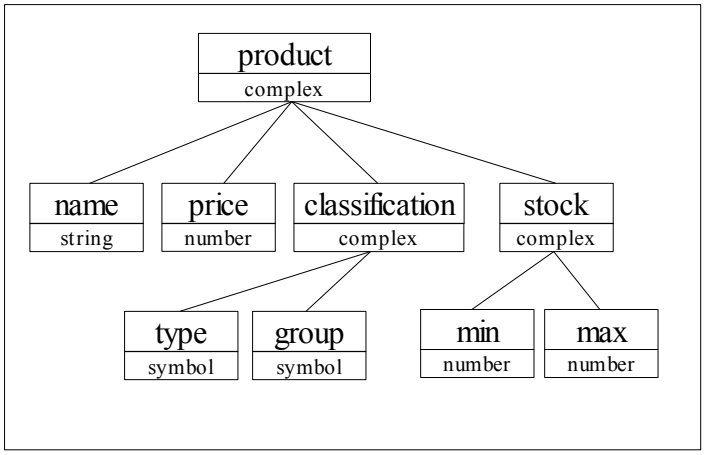


*Figure 20: The tree structured product data*

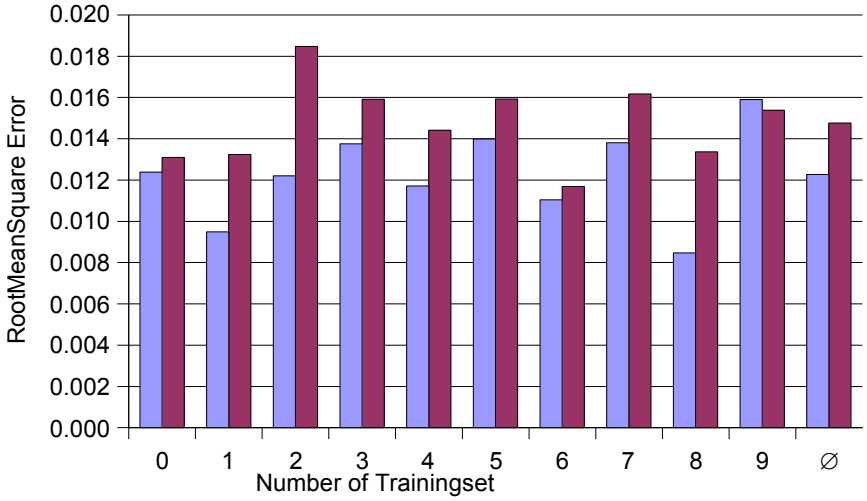


*Figure 21: Results comparing tree structured data*

**Using slightly different data structures**

This experiment shall show whether its also possible to learn measures that compare two objects which have not exactly the same structure. To create such a data set, randomly some attributes in the objects are removed, in this case every attribute of every object in the data set has a chance of 10% to be removed. To calculate the similarity between two objects then following rule is used:

- If an attribute a exists in both objects use the local similarity measure to calculate this (local) similarity value.
- If in one object the attribute is missing set this (local) similarity value to 0.
- If both are missing set the weight for this attribute to 0.

Again a 10-fold cross validation has been done to evaluate if the GA is able to learn good similarity measures that could deal with small differences between the compared objects. Figure 22 shows the results which again are very similar to the results from the second experiment (Figure 19), which is used as reference experiment. In numbers the average root-mean-square error is 0.0131 (while in the reference experiment it was 0.0144). The fact, that in average the root-mean-square error was even a bit smaller is not significant. But significant is, that the results are in the same value range (between 0.01 and 0.02) in opposite to the first experiment where the results are approximately about a factor 6 worse.

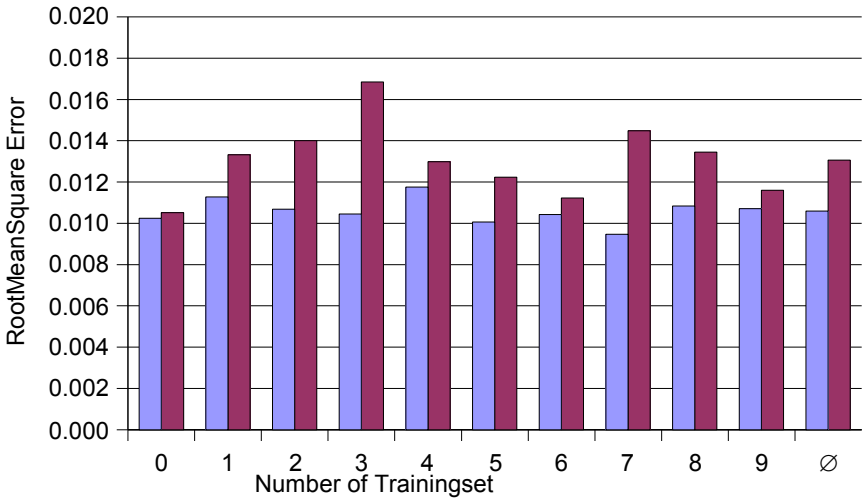


*Figure 22: Results comparing objects with small structural differences*

**Summary**

In this section we compared the cross validation results, and in the most right bars of the diagrams always the average performance of all training- and test set evaluations, that have been done for the particular cross validation, is shown. But if a cross validation is done, it is of course also possible to take the overall best measure (according to the smallest test set error). Figure 23 shows from each

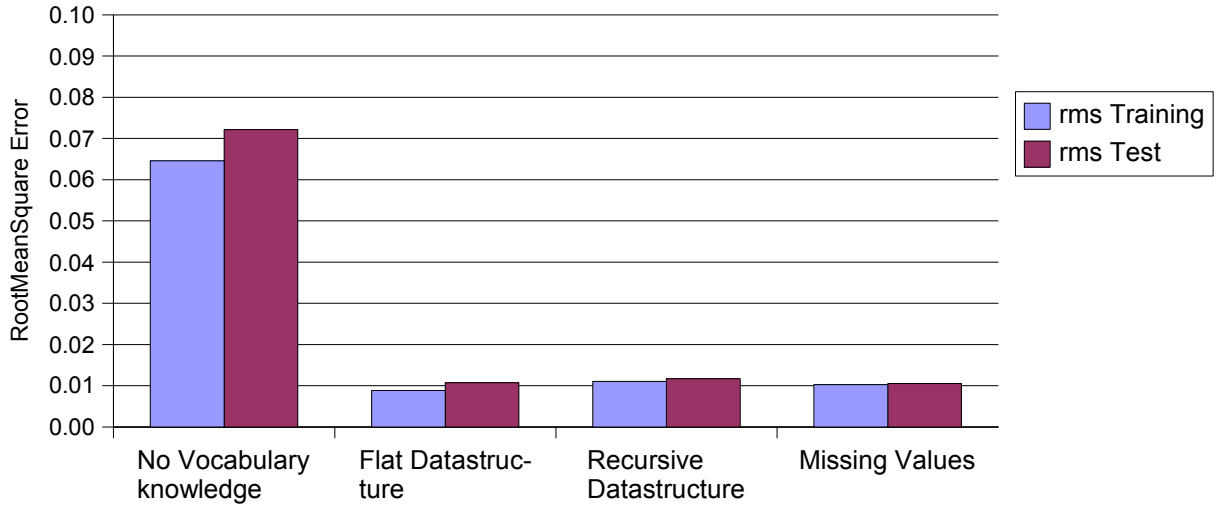


*Figure 23: Summary of the cross validation results (best test set performers)*

experiment the best result that has been generated. So we can see that the framework performs well in all three cases where vocabulary knowledge is exploited. It would also work without vocabulary knowledge but one would have to wait very long for usable results (due to the much larger search space, the GA needs bigger populations and more generations to come up with good results).

## 5.2. Ontology Alignment

In this second group of experiments the goal is to find a alignment between different ontologies. An alignment is needed, to know whether in two different ontologies there are classes or properties defined which actually represent the same concept. Moreover it will show which of them represent the same concept. The problem is, that in different ontologies the same concepts could be named differently (e.g. if the ontologies are written in different languages) and located in a different position in the taxonomy.

### 5.2.1. Task Description

In these experiments the goal is to find the properties, that represent the same “real world” property of an object in two different ontologies. Having such an alignment, the concrete instances of the two ontologies can be merged together. The idea is to do that using a similarity measure that compares properties from the two ontologies. If the calculated similarity value is greater then a defined threshold it is assumed that they are semantically identical. To do so, a reference alignment is needed, that consists of property pairs which match. This reference alignment than will be used to set up a gold standard and so to learn these similarity measures.

### 5.2.2. Converting the Ontology into a Set of assimilable Instances

The first step to do this, is to load the ontology and extract all properties that are defined in it. Then a suitable data structure, that represents the properties, has to be found and the properties' (meta-) data has to be filled into it. This could be, in the most simple case, just the name of the property. In a slightly more complex case, the class names of the domain and the range of the property could be used also, which then leads to a flat data structure suitable for a local-global similarity measure. In a next step the class hierarchy of the ontology could be generated. Having this information for the range and domain classes of each property, information about the class' position in the class hierarchy can also be added to the data structure. This can lead to the data structure shown in Figure 24, and so a recursive similarity measure can be used to compare in this way structured property (meta-)data. For the experiments the ontologies from the “EON Ontology Alignment

Contest" [Euzenat *et al.* 2004] are used.

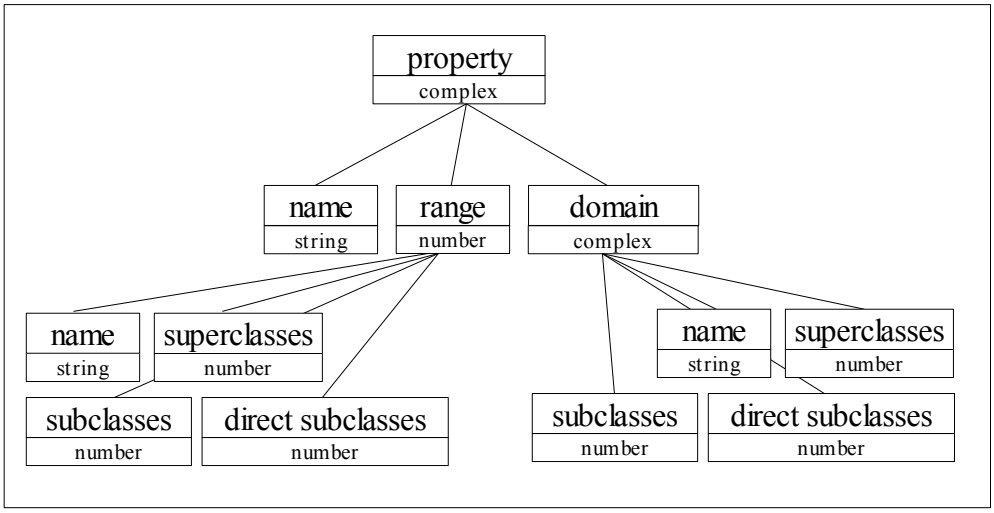


*Figure 24: A possible data structure for ontology properties*

### 5.2.3. Converting the rdf-Reference into a Gold Standard

To create a gold standard, the reference alignment is needed. These reference alignments are also available at the web page of the "EON Ontology Alignment Contest" [eon 2004]. This is a rdf-file but it actually contains a plain XML-data structure. Reading this would lead to a set of property pairs which match. A gold standard can be set up by initialising a matrix and then set the value of each cell to 1 if the corresponding property pair is part of the reference alignment. All other values are set to 0.

### 5.2.4. Set-up of the Experiments

The reference alignments are always defined between the "original" ontology (number 101) and another one. So in the following experiments it will always be tried to find the alignment between the original and another ontology. Following experiments have been done: First an experiment that tests the concept of training- and test set for this group of experiments. This is done for the ontology that contains different naming conventions for the properties and classes (number 204). Because the amount of data is rather small, there are 44 different properties defined. This is done using a 2-fold cross validation. This experiment uses the ontology number 204 (naming conventions). Afterwards experiments using different data structures for the properties are done. Each of these experiments is done for the ontology with different naming conventions, for one using synonyms (number 205) and one with names in another language (number 206).

## 5.2.5. Results

**2-fold cross validation**

As the cross validation results shown in Figure 25 one can see that also using a training set, that contains only 22 properties, no specialisation (over-fitting) can be observed. Actually the five not found alignments will also be missing using the whole property set as training set. The reason for this is that these properties are lower case in one and upper case in the other ontology. This leads to a maximum Levensthein distance, but it would be no problem to just use a case insensitive version of the Levensthein algorithm, or better letting the measure learn whether it should work case sensitive or not. As performance measure for the figures, all correct found alignments minus all wrongly found alignments is taken, so that in this case the maximum would be 22.


*Figure 25: 2-fold cross validation for ontology alignment, Performance is defined as correctly identified alignment minus wrongly identified ones.*

**Using only property names**

Using only the property names will lead to a quite good performance at the case where the alignment for the ontology number 204 (naming conventions) is created, but rather poor results for the other two cases. Note that in all these experiments no 2-fold cross validation is done but all 44 properties are part of the training set. So the maximum which the performance measure can reach is 44. However in the case of the ontology with the foreign names it performs slightly better than the one with the synonyms.

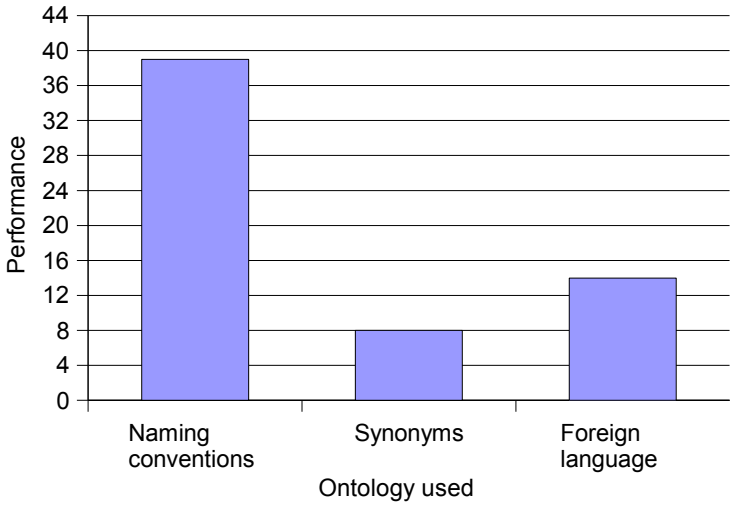


*Figure 26: Training set performance using property name only*

**Using property names and class names of the property's range and domain**

In this experiment the alignment to the naming conventions has the same performance as using the property names only (actually it is even slightly worse). The reason for this is that the names just vary so little that no more improvements can be done using more data. The case with the synonyms also shows no improvement at all but the alignment to the foreign language gets a bit better.

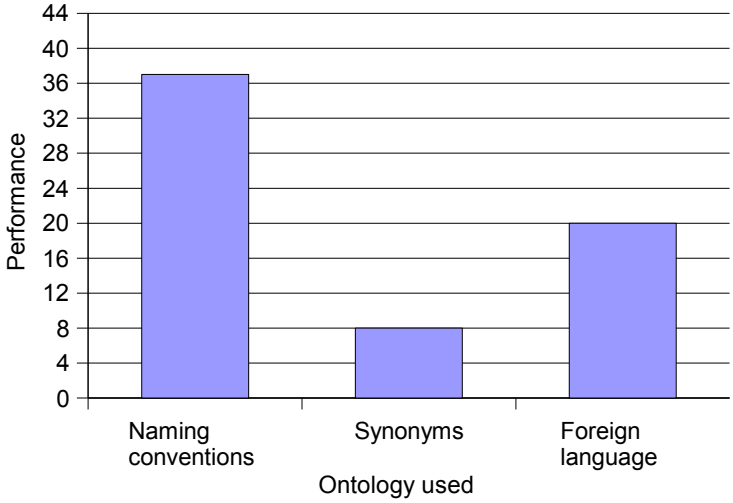


*Figure 27: Training set performance using property name and class name of domain and range*

**Using property names, class names and class hierarchy of domain and range**

In this case also information about the class's position in the class hierarchy is used to find a good alignment between the ontologies. The idea of doing so is, that maybe names of the properties and the class names of their range and domain can be far from similar, but the position of the classes (of domain and range) in the class hierarchy may be the same or at least very similar. For this actually 3 numerical values are calculated and exploited by a numerical (local) similarity measure. These are number of superclasses, number of subclasses and number of direct subclasses. As Figure 28 shows, this finally brings a even worse result to the alignment to the synonym's ontology. Also the alignment to the foreign name's ontology is again as bad as it was using the property names only. The problem doing so is that the class hierarchy in this case is rather small and flat so that most classes have the same amount of superclasses and 0 subclasses. Actually the results are so bad because a lot of

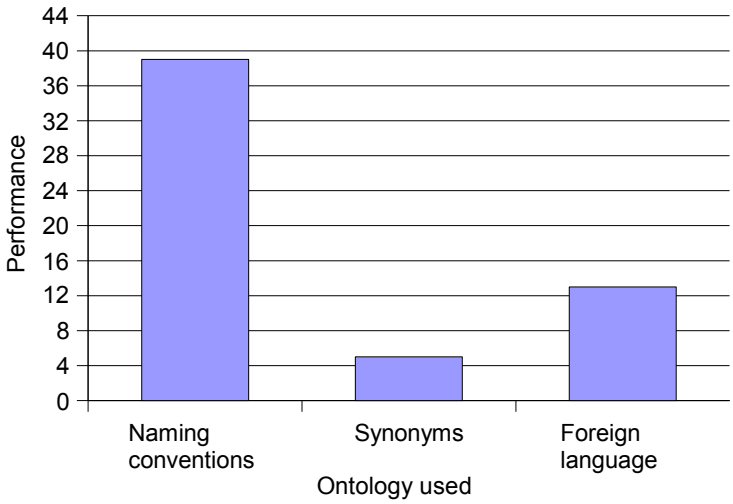


*Figure 28: Training set performance using "position in hierarchy"*

wrong alignments have been found. This leads to the fact that the learning algorithm will set the weights for the similarities of the "position in hierarchy" attributes very low. So again a measure that overweights the name of the property will come out. Note the similarity between Figure 28 and Figure 26.

**Using the position in hierarchy as string attribute.**

Another way to exploit the "position in hierarchy" data is to encode the three numerical values into a string. For this string attribute the another string similarity measure is learned. The benefit of doing so is that also one different value leads to a different string, unlike in the last case where 2 out of the 3 of the attributes will still have maximum similarity. In fact this leads to better alignments as Figure 29 shows.

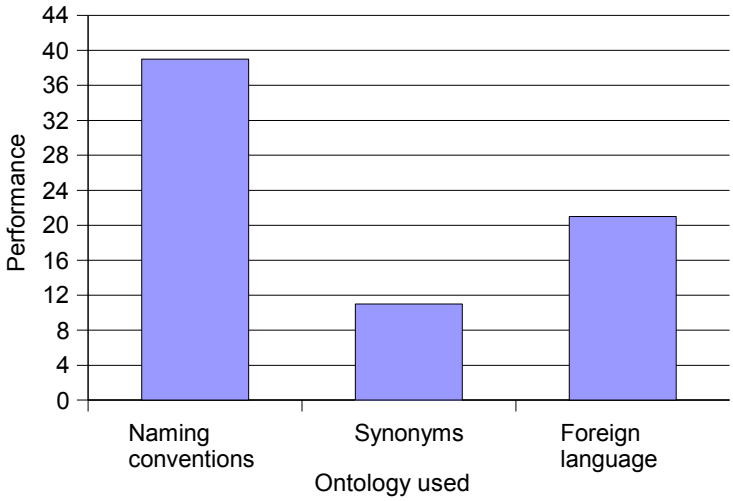


*Figure 29: Training set performance using "position in hierarchy" encoded as string*

## 5.2.6. Discussion, Future work

Looking at these experiment's results, they are, except the easy case with the naming conventions, quite dispiriting. So we need to think why this is so and how further improvements could be realized.

**Using larger ontologies**

In larger ontologies (at best with larger class hierarchies) the chance that two classes which are not semantically identical, but nevertheless have the same values for superclasses, subclasses and direct subclasses will be lower. As mentioned in the explanation of the third experiment (the first one using the class hierarchy), this is a reason that these values don't really help finding the correct alignment. Therefore I predict better results using just a larger ontology.

**Using even more property (meta-)data**

More meta data of the properties can be taken into account, e.g. the cardinality could be exploited also. But this would have the same problem as the "position in hierarchy". The value of this attribute would in most of the cases just be 1. So for a lot of property pairs, which are not part of the correct alignment, the comparison of this attribute will lead to a maximum similarity.

**Using a class alignment**

The basic idea of using the class hierarchy was to identify the classes that represent each other semantically, and so reasoning, that properties where the range and the domain is the (semantically) same class have a good chance to represent each other. Now the same procedure, as is used finding the alignment between the properties, can be used for finding the class alignment. Having this done, it can be exploited to find the property alignment.

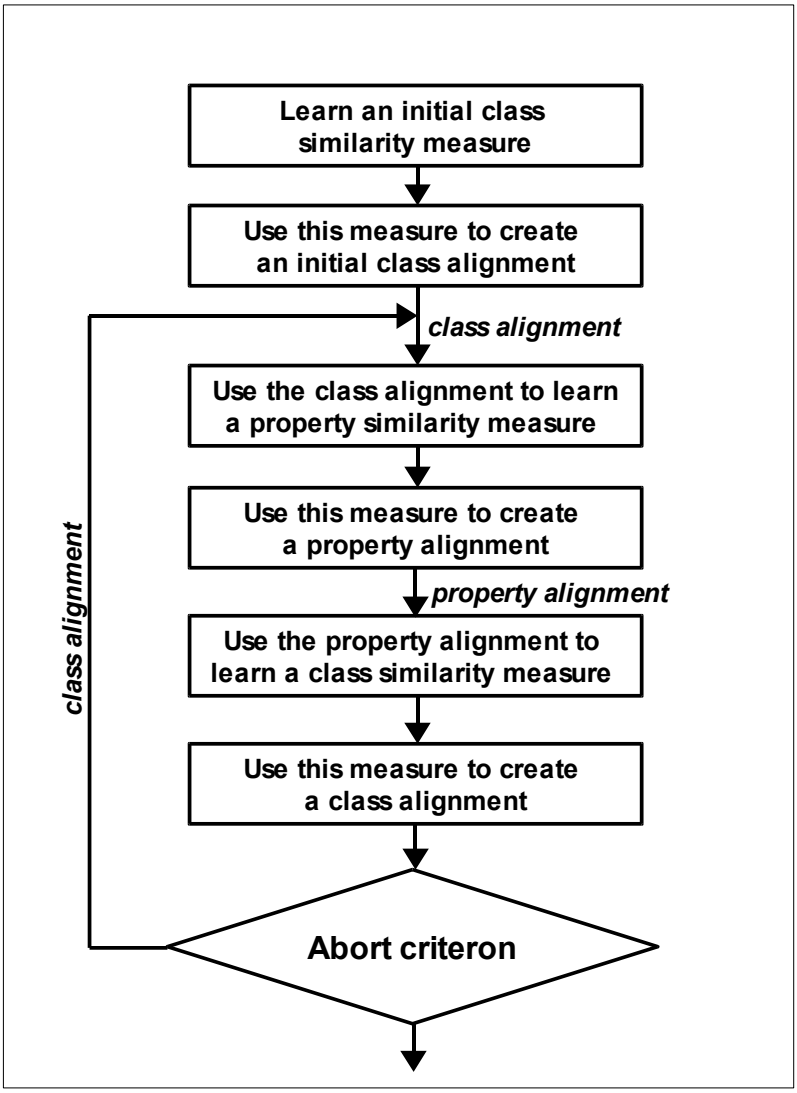


*Figure 30: Suggested learning cycle for an ontology alignment*

**Using an alignment learning cycle**

As mentioned before, the class alignment could be learned in a first step analogue to the the property alignment. Looking at the (quite poor) results finding the correct property alignment, there is little hope that finding the class alignment would perform much better. But to find the class alignment, the property alignment could of course also be exploited. So this could be done in a cycle, an initially quite bad class alignment leads to a slightly better property alignment. Than this again is used to learn a better class alignment and so on and on. Figure 30

shows the imagined learning cycle. Note that in the Figure it is the class similarity measure that is learned initially and therefore an initial class-alignment will be created. This is arbitrarily defined, one can also define to start the cycle with an initial property alignment.

# 6. Genetic Programming (GP)

So far we have seen, that it is possible to use the principle of evolution to learn meaningful similarity measures. To do this, the genetic algorithm, described in Chapter 4, has been used. Using a GA, all parameters of a solution (i.e. the similarity algorithm) have been encoded to a genotype (i.e. the chromosome). Another domain specific operation was used to decode the chromosome and so to build up the concrete (working) solution. Another way of exploiting the mechanisms of evolution, is to use genetic programming instead [Koza 1992]. This method generates directly the solution (since the solution is a computer program), therefore the en- and decoding parameters to and from a chromosome are not necessary any more.

## 6.1. Introduction

The idea of genetic programming is, that instead of using a sequential chromosome to reproduce individuals. A tree data structure is used for this. And since every computer program can be represented as a tree, exactly these trees are taken to do the reproduction. Figure 31 shows how a program, which calculates the hypotenuse of right-angled triangle, can be represented as a tree. Having such a tree as "chromosome" for the genetic programming, no (domain specific) interpretation is needed to get a working program. Of course the meaning of the symbols has to be interpreted, but this has to be done anyway while compiling the program This interpretation stays always the same for the same programming language. Having this, a learning cycle similar to the one in the genetic algorithm can start. First a fitness function evaluates each program's fitness, and according to these fitness values a selection is performed. The genetic operator works a bit

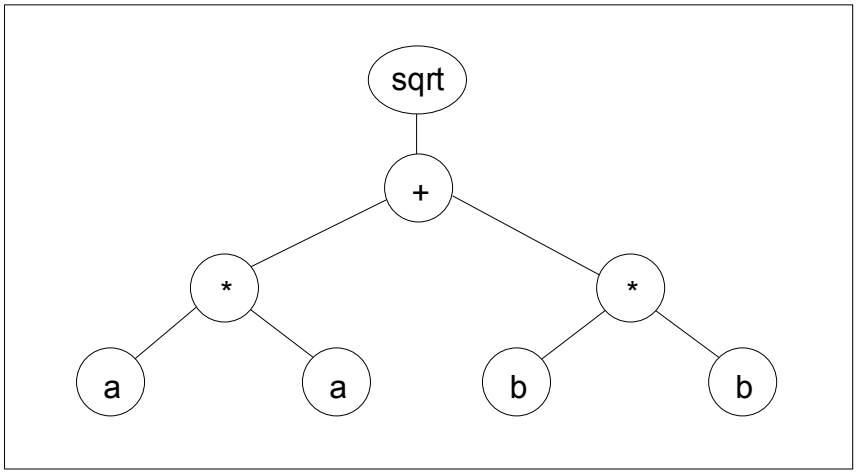

*Figure 31: Tree representation of the Pythagoras' theorem*

different, but in principal has the same goal, namely, to create a new generation using the selected programs as parents. The details of these steps and the differences to the classic GA will be described in the section “6.3. Genetic programming in Detail“ later on in this chapter. But genetic programming needs also some solution specific configuration. Namely, a terminal- and a function set has to be defined. In the example in Figure 31, the terminal set would be the two variables a and b, the function set would be {sqrt, +, *}. Elements of the terminal set are candidates for the leaf nodes of the tree, the elements of the function set are candidates for the inner nodes. Additionally for each function its arity has to be defined, in our example the arity is 1 for the square root function and 2 for the addition and the multiplication. This value defines the number of child nodes, which a node, representing a particular function, must have.

# 6.2. Conditions for successful genetic programming

In this section two conditions that have to be fulfilled for a successful genetic programming in practice will be presented.

## 6.2.1. The Closure Property

This property needs to be fulfilled, to achieve that every possible tree, constructed out of the function set and the terminal set, will represent a working program. For this, it is necessary, that every function (in the function set) accepts every possible return value of any other function at every position in its argument list as input parameter. This leads in practice to the fact that all functions have to return the same data type, and also that all parameters of each function have to accept this data type (or class in an object oriented programming language). Moreover all possible values, which the terminals might have, must be of this data type. Otherwise a randomly created tree will mostly represent a program that will not work due to type errors.

Also each function's return value has to be defined for each combination of input value that may occur. In the example the square-root function must not be the native square-root function, which is usually not defined for negative input values, but must be replaced by a “protected” version. This version must also return a result for negative input values. One way to do this, is to define sqrt as follows : $sqrt(x)=\sqrt{|(x)|}$ . For the division function a similar workaround to avoid the “division by zero” error has to be done.

### 6.2.2. Sufficiency of Terminal- & Function-set

Another mandatory condition is, that the terminal- and the function set have to be sufficient to create a program that can solve the problem. In our example genetic programming would never have the chance to come up with a proper solution, if for example the terminal b (representing the length of one leg of the right-angled triangle) would not be part of the terminal set. Also, if any of the functions would be left away, it would not be possible to get a good solution. On the other hand, it is possible to define more functions and terminals than are actually used (since we want to use genetic programming to generate a solution, which we do not know in the first place). Therefore knowing that the problem of calculating the length of the hypotenuse is a mathematical problem and depends on the length of the legs of the right-angled triangle, one could define a terminal set {a,b} and a function set {+, - , *, /, sqrt, sin, cos). So we have seen that some knowledge about the solution's character is needed also in genetic programming.

## 6.3. Genetic programming in Detail

Analogue to chapter 4 about the GA, some details of the single steps in genetic programming will be presented in this section. They are more or less the same as in the genetic algorithm so especially the differences will be mentioned. Also the learning cycle as a whole is quite the same as in GA (see Figure 32).

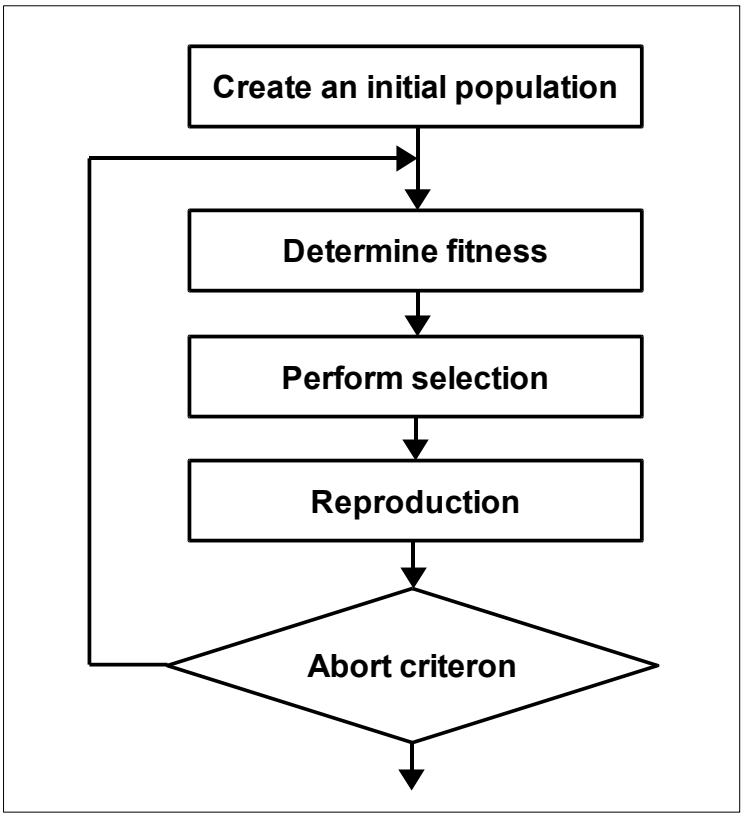


*Figure 32: Flowchart of the learning cycle in GP*

### 6.3.1. Creating an initial Population

In genetic programming the very first step is also to randomly generate an initial population of individuals. But since the data structure, representing an individual, is a tree in this case, there are some differences. Opposite to just setting a random value for each gene in each chromosome, a more sophisticated method to build up the initial population is needed. Basically two methods exist

to randomly generate a tree, consisting of the terminals and functions given in the corresponding sets. Koza calls these methods the "full" and the "grow".

The first (full) method is to define a desired height (or number of levels) of the tree (e.g. 5). For the root node a symbol of the function set is randomly chosen. The number of needed child nodes is defined by the arity of the function which is chosen. The required number of child-nodes will be generated and a symbol from the function set is set randomly to each node. These steps are then recursively repeated for each node until the desired height of the tree is reached. In this case for each child needed, a symbol from the terminal set is randomly chosen to create the leaves of the tree. This method leads to trees, where each leave is on the same level (i.e. the desired height of the tree). This is then a "full" tree.

The "grow" method is (except for the root node) to choose always a symbol out of the union of function- and terminal set. If for a particular child a terminal symbol is chosen, this branch ends there. Additionally a maximum height of the tree must be defined. If this height is reached, choosing a terminal symbol will be enforced. This is has to be done, to avoid that a particular tree grows infinitely large. Trees created with this method can have branches of different lengths.

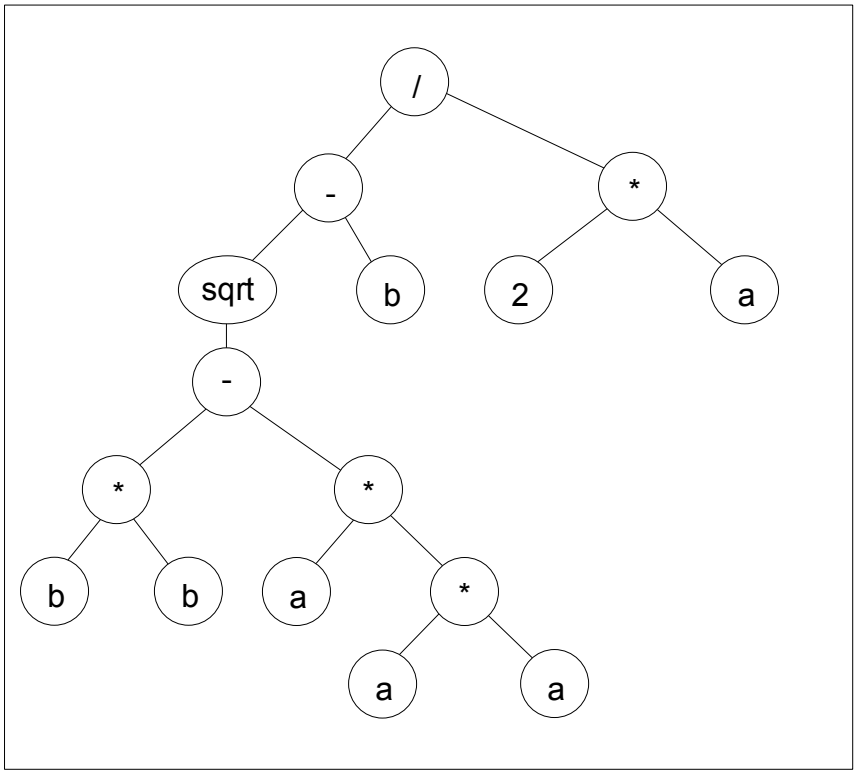


*Figure 33: A tree with branches of different lengths.*

The tree shown in the Pythagoras example is an example for a "full" tree. All terminals are on level 3, if counting starts with 0 at the root node (which is the method in this diploma theses). Figure 33 shows a tree that could have been generated with the "grow" method. Maybe the maximum level was 6, but this is not necessarily the case because the process of growing can stop before the maximum level is reached. This tree actually calculates the first solution of the a quadratic equation $ax^2+bx+c=0$ .

To get a good diversity in the initial population, one can use the following strategy: Half of the population is created with the full method, the other half with the grow method. These two segments again are divided into *n* parts (e.g. $n = 5$). Then the maximum number of levels for part *n* of each segment is defined as *minHeight* + *n*, whereby *minHeight* is e.g. 2. Doing so, one gets a good diversity of trees of different sizes. For each size some "full" trees and some with variable branch lengths, will be part of the initial population. Koza calls this method "ramped-half-half".

### 6.3.2. Fitness function in genetic programming

The fitness function in GP actually is very similar to the one in GA. All considerations done in section "4.3.1. Fitness Calculation" (see page 22), and especially the transformation from deviation to fitness (in the case of learning similarity measures), are still valid. As also mentioned in chapter 4, a difference is, that sometimes the first step of the fitness function is to build up the phenotype from the genotype (interpreting the chromosome). This is not necessary in GP because the learned structure is the program itself. In practise a function will be executed (e.g. "evaluateTree") which takes the learned tree as argument and returns the calculated result. If a programming language is used, which is able to interpret/compile and execute its own source code at runtime, it is possible to create the source code from the tree and then letting this code snippet being executed.

### 6.3.3. Selection

As the fitness calculation, the selection step is also very similar to the analogue step in the GA. All considerations from section "4.3.2.4. Combination of the Strategies" (see page 25) are valid. It will also be possible to basically use a roulette wheel strategy. To enrich this strategy an elite can be defined, in the meaning that members of the elite will have a guarantee to be selected once or *n*-times (where *n* would be the guaranteed reproduction rate). Furthermore the selection can be combined with a defined amount of individuals which are selected just randomly, regardless of their fitness values. One might consider to use different parameter values, but as also already mentioned, the algorithm is quite stable against different parameter settings, and so the expected impact of doing so will be small.

### 6.3.4. Genetic Operator

As the representation of the individuals is different to the GA, also the genetic operators work different. The basic principle of crossover and mutation remains the same, but the method to achieve them is different. Crossover in GP is also done by first selecting the two individuals which

participate (the parents). Having selected the two parents, two crossover points (unlike one only in GA) are randomly selected (the grey nodes in Figure 34). The crossover points may be any inner- or leaf node of the trees. The two selected nodes (including their child nodes) then will be exchanged between the two trees, and so the offspring is generated (shown in Figure 35). If in both trees a leaf node is chosen to be crossover point, the operation has the same effect as a mutation in GA (one node in each tree will change its value). Therefore it is not necessary to have an explicit mutation operator in genetic programming (but one could program and use a mutation operator if desired). To achieve that (hopefully meaningful) program parts will be exchanged instead of leaf nodes, one can define that in e.g. 90% of all cases an inner node gets selected as crossover point, and on the other hand only with a chance of 10% a

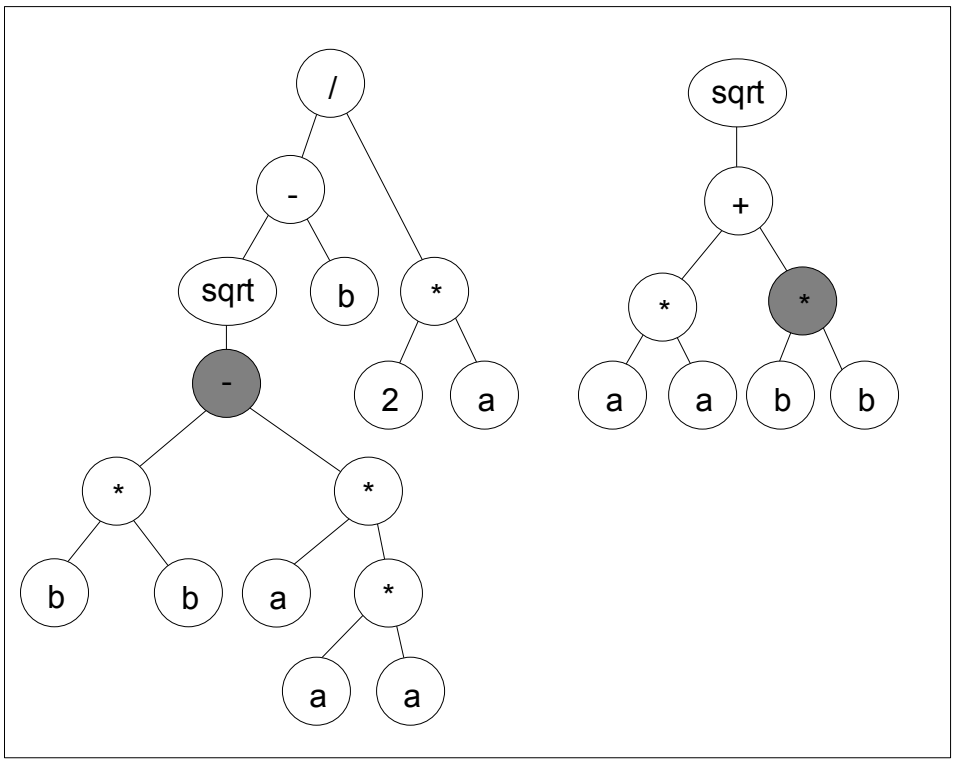


*Figure 34: Two inner nodes selected for the crossover operation*

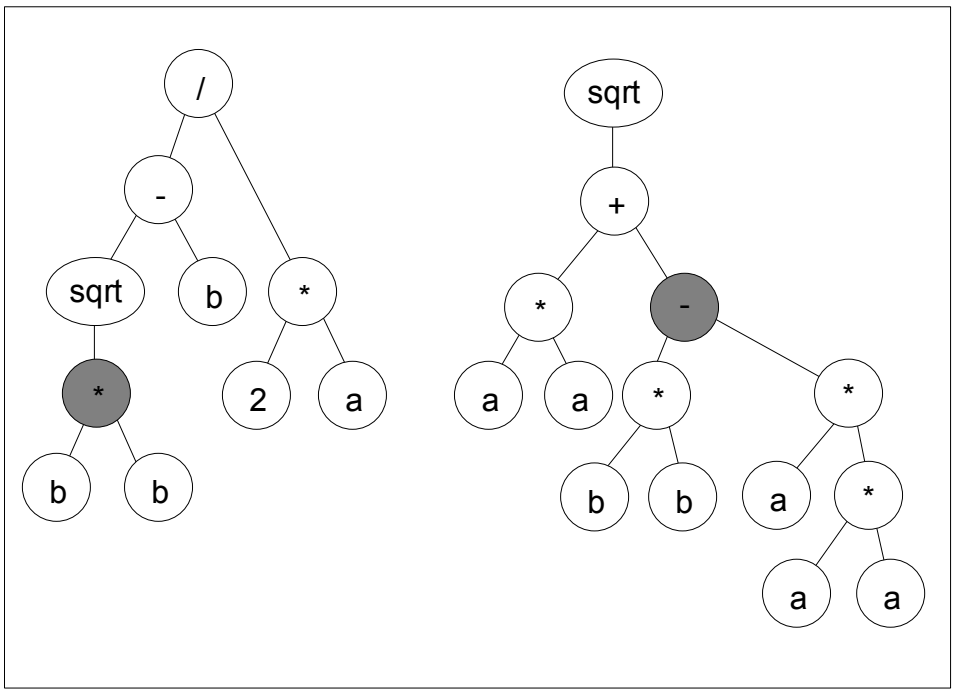


*Figure 35: The offspring of the crossover operation*

leaf node will be selected (independent of the inner-node – leaf-node ratio in the tree). Unlike in the genetic algorithm where the length of the chromosome is fix, the trees here may grow and shrink. In the example of Figures 34 & 35 the left tree shrinked and the right one grew. The reason for this effect is that the two sub-trees, which have been replaced, did not have the same size. Therefore it is necessary to define a maximum size which a tree may reach, otherwise the trees may grow infinitely large and so cause an unacceptable runtime.

# 7. Results using GA & GP

In this chapter we will show how genetic programming can be used to learn similarity measures. For that several evaluations will be presented. The goal is to find out if, and under which conditions, genetic programming can successfully be used as learning method to find good similarity measures.

## 7.1. Set-up of the Experiments

Due to the closure property (see section "6.2.1. The Closure Property", page 48) it will not be possible to learn local similarity measures because (except numeric similarity measures) they have different input and return types (e.g. Levensthein takes two strings and return a double value). Therefore learning the local measures will, also in these evaluations, still be done using the GA. Genetic programming will be used to learn the amalgamation function (i.e. the function that combines each local similarity value to a global similarity). This amalgamation function was fixed to be a weighted average in all experiments using the GA only, and the single weights then have been learned with the GA. Now the amalgamation function will be a learned genetic program.
So in all following experiments first the normal GA runs and learns a global similarity measure and afterwards a genetic program is learned. This genetic program will get the local similarity values (calculated by the local similarity measures of which the (GA-learned) global similarity measures consist).
Figure 36 shows two different learning rates of two different runs, learning an amalgamation function for a global similarity measure. On the y-axis one can see the achieved root-mean-square error using the particular amalgamation function that is learned. The detailed conditions of the runs are not relevant here. What should be shown with this figure is, that the learning rate in genetic programming may have quite different characteristics from run to run. In run 1 the amalgamation function first gets even worse and afterwards makes a sort of stepwise improvements. In run 2 the root-mean-square error shrinks quickly after having started the learning cycle, and already after generation 40 the root-mean-square error reaches almost the final value.

As in the evaluation of the learning with the genetic algorithm only, in the next two sections results using the product dataset and results of trying to find alignments between properties of different ontologies will be presented.

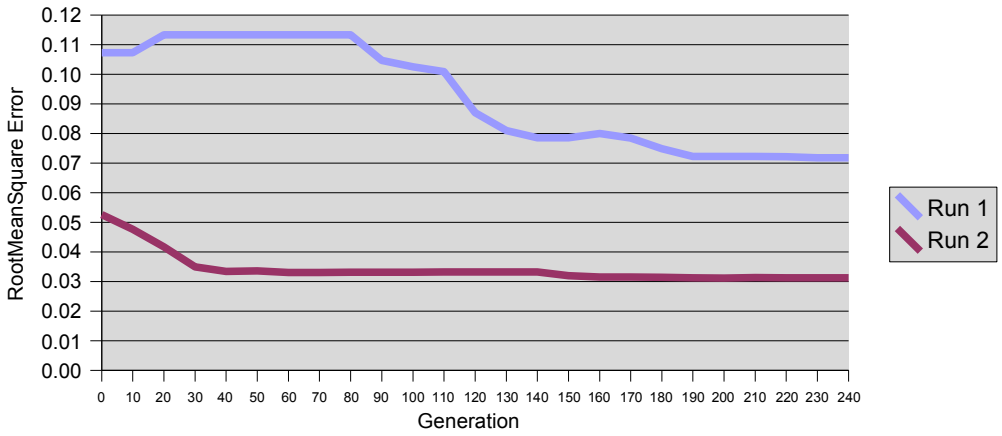


*Figure 36: Two learning rates of the genetic programming*

Because usually the search space is much larger in genetic programming than it is using a GA, genetic programming needs larger populations, to come up with good results, than the genetic algorithm[1]. For that the population size in all experiments presented is 500 for the genetic programming part.

## 7.2. Results using the Product Dataset

In this section the results of four experiments will be shown. For one experiment a 10-fold cross validation has been done. For the other three only a training- and test set evaluation will be presented (due to the fact that a cross validation using GA and GP uses about 10 hours of runtime on a 1.6 GHz AMD processor).

So what are the four experiments? As mentioned in the last section the procedure is to learn a similarity measure with the GA, and then use the so learned local similarity measures as given, trying to find a good amalgamation function. So one can define whether the GA may learn the weights ($\omega_1, \omega_2, \omega_3, ..., \omega_n$) of its own amalgamation function (the weighted average),

$$\text{Sim} = \sum_{i=1}^{n} \omega_i \cdot localSim_i$$

or whether these aspects will not be learned in the GA part of the experiment. In this case, the amalgamation function the GA may use, is fixed to be a normal (equal weighted) average,

1 Due to this fact detractors of genetic programming say that it is more like random search than really exploits the principals of evolution.

$$Sim = \sum_{i=1}^{n} \omega \cdot localSim_i$$

while $\omega$ is fixed to be $\frac{1}{n}$ . Independent of this decision, another option is to transform the amalgamation function, which the GA uses (or has learned), to a genetic program. This program then can be added (or not) to the initial population, that the genetic program part of the experiment uses. An amalgamation function that combines for example four local similarity values,

$$Sim = \sum_{i=1}^{4} \omega_i \cdot localSim_i$$

can be transformed in the genetic program shown in Figure 37. This transformation of course can be made independently whether the weights in the GA were all fixed to be $\frac{1}{4}$ or have been learned by the GA. So these two options lead to four combinations shown in Table 6 (where AF stands for "amalgamation function") and the results using these settings will be presented.

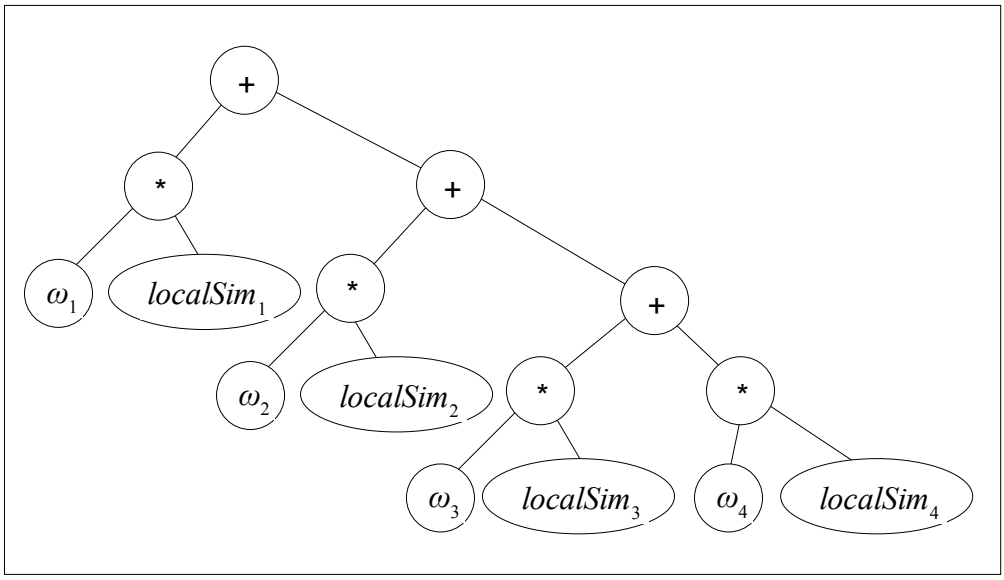


*Figure 37: An amalgamation function transformed to a parse tree for a genetic program*

| | GP does **not** use GA's AF | GP uses GA's AF |
|---|---|---|
| GA may **not** learn AF | Experiment 2 | Experiment 1 |
| GA may learn an AF | Experiment 3 | Experiment 4 |

*Table 6: Table of the following experiments*

**Experiment 1**

Figure 38 shows the results of the 10-fold cross validation. In this experiment the genetic algorithm was fixed to use an equal weighted average as its own amalgamation function and this equal average function has been transformed to a genetic program. This genetic program was than added to the initial population of the genetic algorithm. The x-axis shows which segment (since in a 10-fold

cross validation, the whole dataset is divided into 10 segments) was actually the training set. All other sections were part of the test set. For each training set used, the first two bars show the training- and test set error which the measure had after the run of the GA. The second two bars show the deviations that have been achieved after the genetic programming part of the experiment. As one can see, in all 10 cases the two first bars, representing the training- and test set error the GA came up with, are higher than the second two bars. These second two bars represent training- and test set error which the measure had after the genetic programming part of the experiment. So we can see that GP brings a benefit in this 10-fold cross validation.

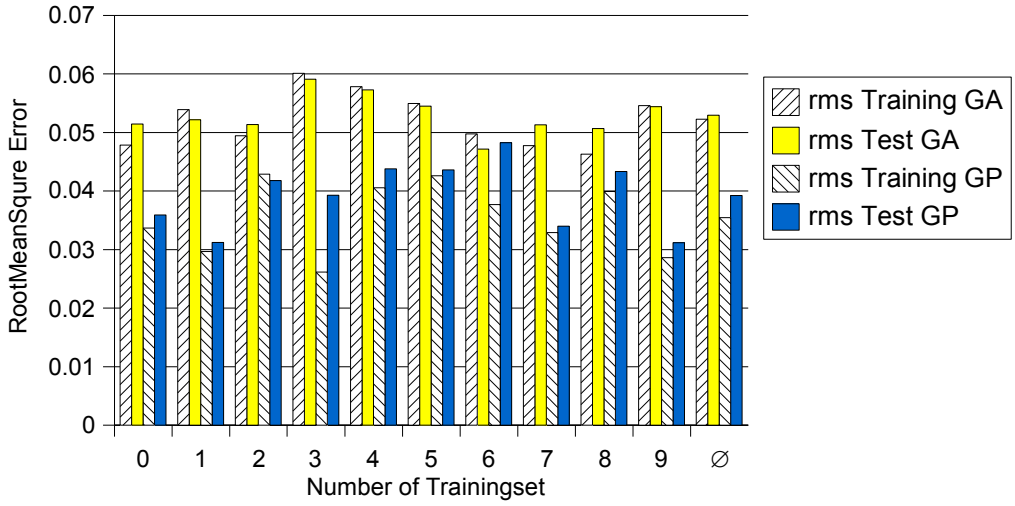


*Figure 38: 10-fold cross validation using GA and GP*

**Experiment 2**

Figure 39 shows the results of the experiment where the GA also had to learn a measure using an equal weighted average as amalgamation function, but this function was not put into the initial population for the genetic programming part.

Looking at these two figures (38 and 39) one can see that the performance of the genetic programming part is quite similar. In both experiments the GA had no possibility to create an

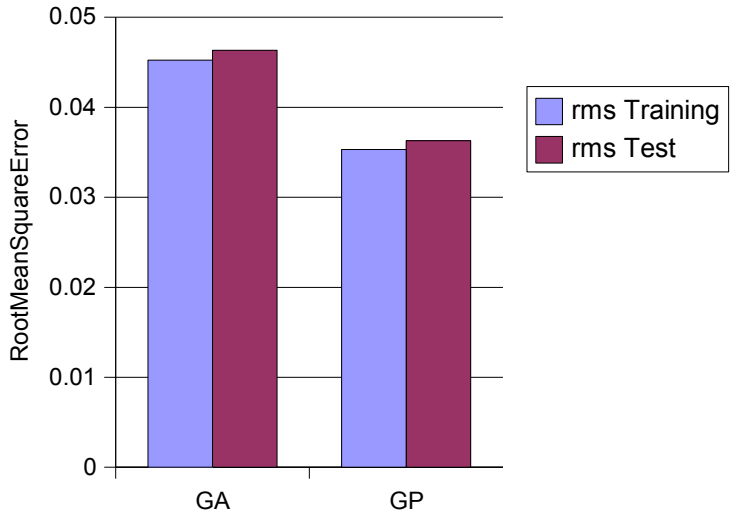


*Figure 39: Training- and test set errors after GA and after GP, the GA's amalgamation function was not part of the initial population of the GP part*

improved weighted average as its own amalgamation function, and so, whether the (simple) equal weighted average function is given to the genetic programming part or not, has almost no impact.

**Experiment 3 & 4**

The next two experiments show the cases where the genetic algorithm had the possibility to come up with a meaningful weighing of the different attributes of the product objects (which are the compared objects in this section). Due to this, at the end of the GA part of the experiment, an improved amalgamation function is contained in the measure which the genetic algorithm learned. Now this amalgamation function again can be transformed into a genetic program. This can afterwards be put into the initial population for the genetic programming part or not (analogue to the last two experiments). Looking at the Figures 40 and 41 one can see, that in this case (GA may learn weights) there is a significant impact if the GA's amalgamation function is part of the initial population of the genetic programming part. If not, as shown in Figure 40, the measure learned with GP is worse than the one the GA came up with. On the other hand, the figures show that the genetic programming part results in considerable improvements, if it gets the GA's amalgamation as a kind of "hint" or a good starting point to find a better amalgamation function that the weighted average.

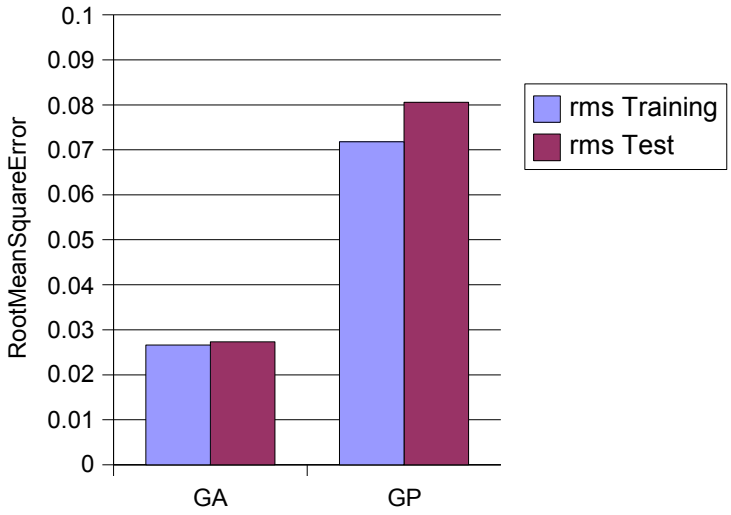


*Figure 40: Training- and test set errors after GA and after GP, the GA's amalgamation function was not part of the initial population of the GP part, but GA had the possibility to learn its own amalgamation function*

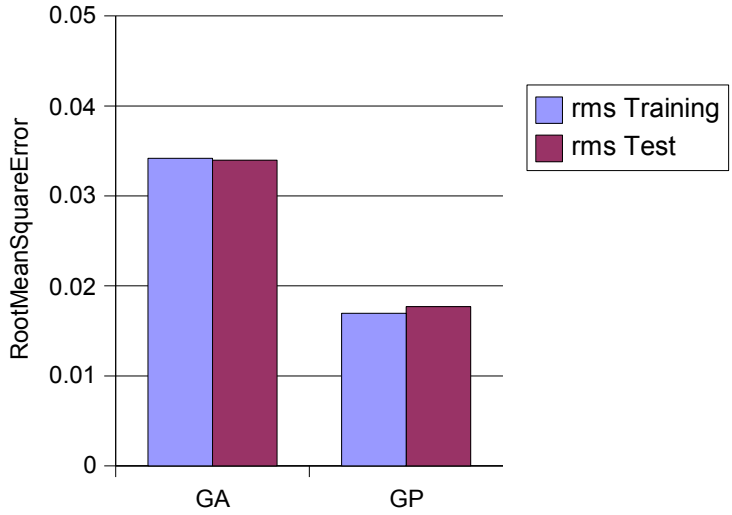


*Figure 41: Training- and test set errors after GA and after GP, the GA's amalgamation function was part of the initial population of the GP part, and GA had the possibility to learn its own amalgamation function*

# 7.3. Results of learning an Ontology Alignment

In this section the results of using the genetic algorithm and genetic programming to find a property alignment between two ontologies will be presented. Generally the task is the same as in the case of using the GA only, as mentioned in section "5.2. Ontology Alignment" (see page 40). As seen, using the GA only, leads, for some ontology alignments, to very poor results. Especially the alignment between the reference ontology and the one, which uses synonyms for class and property names, never came up with a performance better than 11, while the maximum value of the used performance measure was 44 (Fig. 29, page 44). Therefore, in this section (where we will try to improve this by using genetic programming) the alignment between these two ontologies shall be learned in the next two experiments.

As mentioned in the last section, an option of the experiments is whether the genetic algorithm may learn its own amalgamation function or not. As in the previous section an experiment was made for both methods and will be presented. The other option, whether the amalgamation function, which the GA learned, is contained in the initial population, which the genetic programming part gets (or not), is for both experiments set to "yes".

| | GP uses GA's AF |
|---|---|
| GA may learn an AF | Experiment 1 |
| GA may **not** learn AF | Experiment 2 |

*Table 7: Table of the following ontology alignment experiments*

**Experiment 1**

Figure 42 shows the results of the first experiment, where the GA was able to learn its own amalgamation function (the weights of the weighted average). As one can see, also the use of genetic programming does not lead to a perfect (or almost perfect) alignment. But a significant improvement can be observed. The performance, defined as the amount of correctly

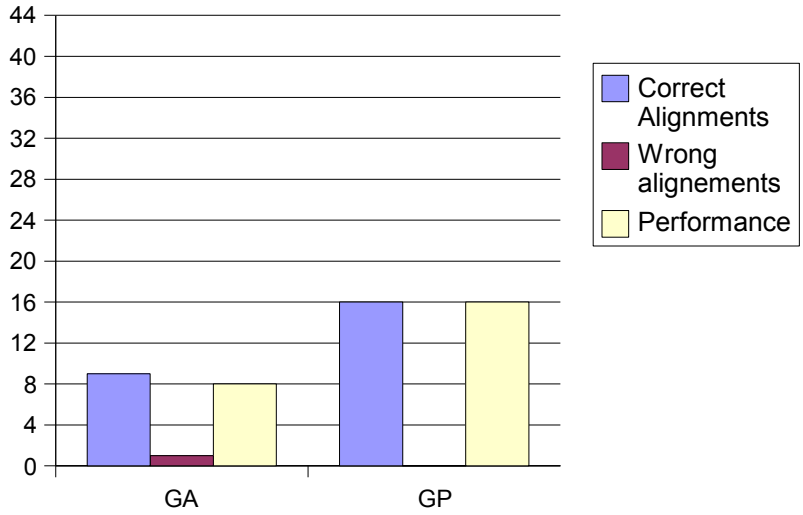


*Figure 42: The correctly- and the wrongly found alignments and the performance as correct ones minus wrong ones*

identified alignments (true positives) minus the wrongly identified ones (false positives), increases in this case from 8 to 16.

**Experiment 2**

In the second experiment the GA was not able to learn its own amalgamation function, and so the similarity measure learned by the GA is very poor, as can be seen in Figure 43. The performance of the measure learned by the GA is even negative, because there are more wrongly identified alignments than correctly identified ones. But looking at the results after the genetic programming part of the experiment, one can see that the performance increased up to 16. This is the same value as achieved in the first experiment. So the results after the genetic programming part does not depend on a meaningful weighting of the similarity measure already after the GA part of the experiment.

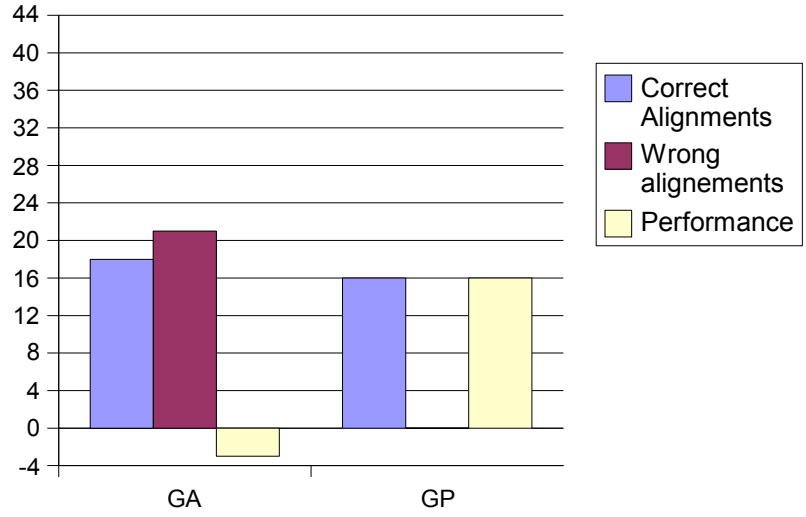


*Figure 43: The correctly- and the wrongly found alignments and the performance as correct ones minus wrong ones, the GA could not learn its amalgamation function*

**Conclusions, Future work**

At the end of the section about finding ontology alignment's using GA only ("5.2.6. Discussion, Future work", page 44), some possibilities to improve the results in future work are presented. One of them is to use a class-alignment to learn the property-alignment and vice versa. Doing so, it is possible to build up a kind of a learning cycle (see Fig. 30, page 45). Looking at the considerable improvements (despite the still quite poor quality of the absolute results) the use of genetic programming can achieve, it would be a good idea, to include the genetic programming part in each learning step of this learning cycle.

# 8. Implementation Aspects

Since this diploma thesis is not just a theoretical thesis about learning similarity measures, but also an implementation in the "real" world has been developed, we will focus in this chapter on some aspects of the implementation of the Local/Global Framework and the learning algorithms.

## 8.1. The Local/Global Framework

The whole implementation is embedded in the so called SimPack library which has been developed at the department of informatics at the University of Zurich. For more information about the SimPack project see [Bernstein *et al.* 2005] and [Bernstein and Kiefer 2006].

### 8.1.1. The Local Similarity Measures

The 4 basic types for Local Similarity measures have all their own class. All these four classes inherit from the abstract class LocalSimilarityMeasure. These five classes can be found in the

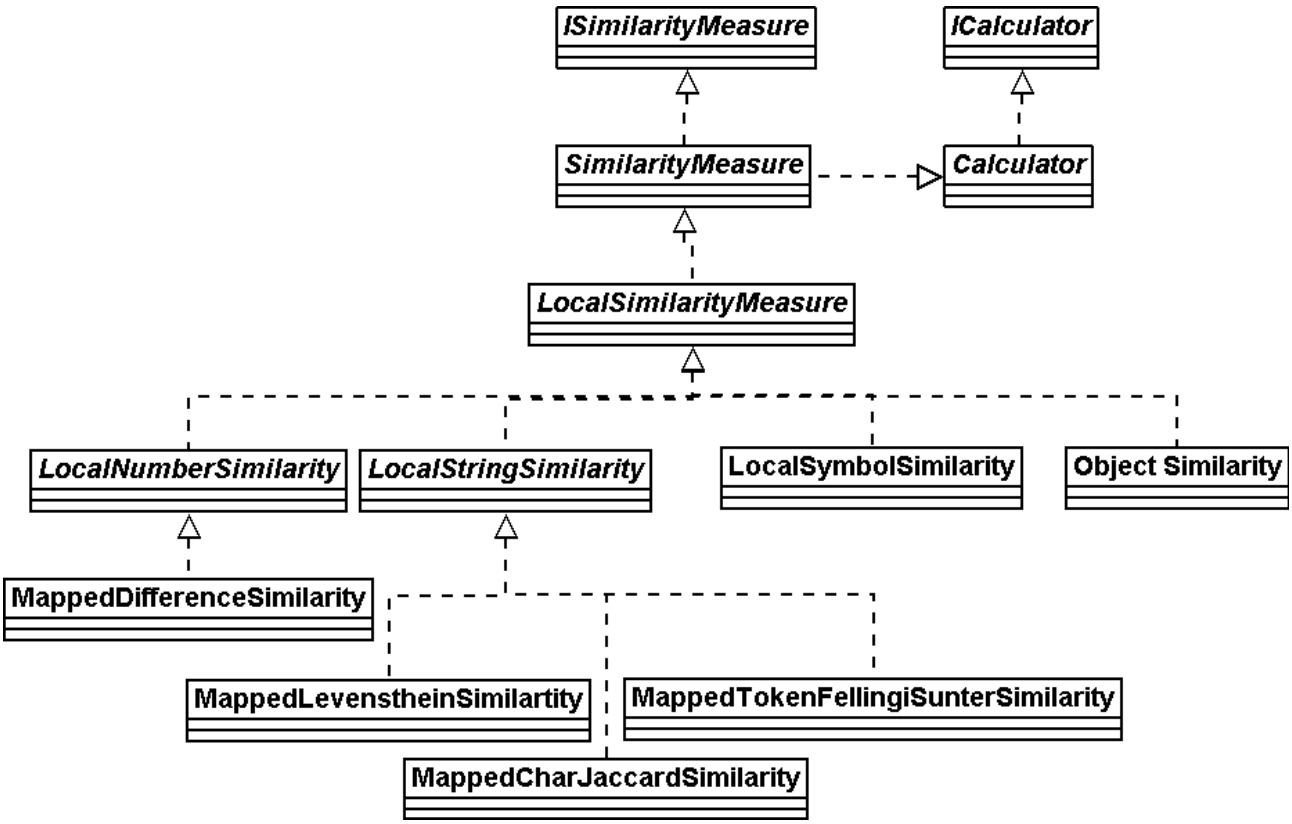


*Figure 44: The class diagram for the local similarity measures*

package `simpack.measure.local`. Because all local similarity measures inherit from one single abstract class, it is easy to define the object similarity class in a way that building up recursive measures will be easily possible (due to the fact that an object similarity measure can be seen also as a local similarity measure). In fact the class ObjectSimilarity contains just an array of LocalSimilarityMeasure, whose entries can be objects of any subclass of LocalSimilarityMeasure. As shown in the class diagram, LocalSimilarityMeasure is a specialisation of the "base" class SimilarityMeasure, defined in the package `simpack.api.impl`. Note the subclasses for the Number and the String measures, they are made to be able to use different internal similarity measures (in the case of String) or distance measures (in the case of Number). More of these internal measures can be added easily.

### 8.1.2. The Configurations

The class diagram of the configurations for the similarity measures is built quite similar as the class diagram for the measures themselves. Basically there exists one configuration class for each type of measure, and all configuration classes implement the interface ILocalSimilarityConfiguration. Note that the configuration for string similarity measures (Sim2SimMapping) is a specialisation of Dist2SimMapping (the configuration for numeric similarity measures). There exists also a class SimilarityTree but it is technically not part of these configuration classes. A SimilarityTree can be used to set the similarity values of a SimilarityTable. This has especially two reasons:

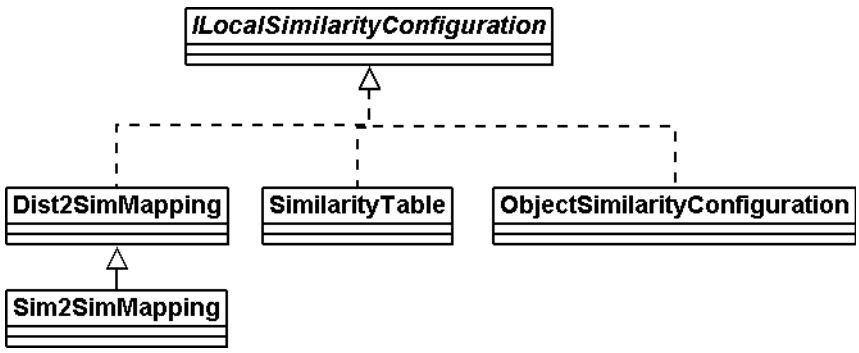


*Figure 45: The class diagram for the Configuration classes*

1. For each similarity measure there is exact one configuration.
2. SimilarityTables are much faster than trees, because in trees we have always to find the nearest-common-parent node (NCP) to get the correct similarity value.

All configuration classes are located in the package `simpack.util.local`.

## 8.2. Evaluation of Similarity Measures

There exists one abstract class that is able to evaluate the soundness of a similarity measure. To be able to use this class also to evaluate how well a learned amalgamation function performs, it is a generic abstract class. Concrete implementations will have to implement basically two methods. One that returns the similarity between two objects which is calculated using the measure to be

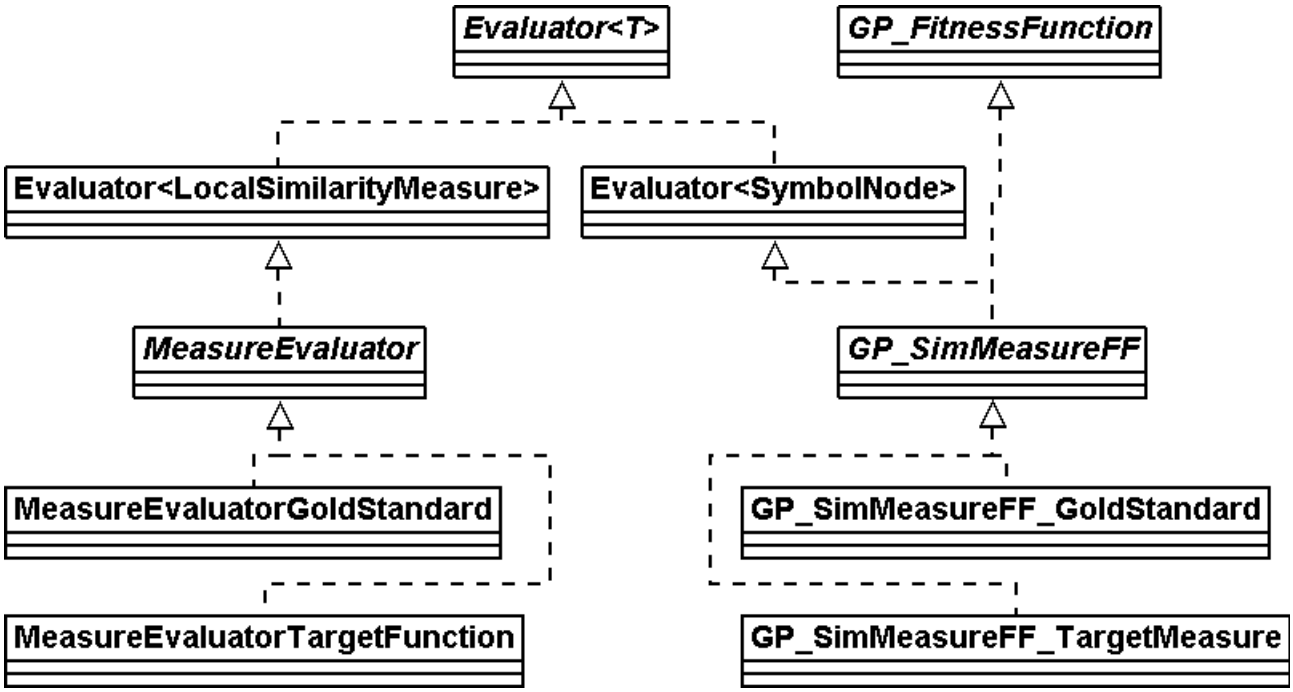


*Figure 46: Class diagram of the Evaluators*

evaluated (or amalgamation function). The other method, which must be implemented, has to return the similarity value between two objects which is defined in the gold standard. The class diagram shows how the different evaluation classes are inherited from each other. The great benefit of having this generic evaluator is that all the code about:

- which deviation type is used.
- how the deviation should be transformed into a fitness value
- how many comparisons are made for one single measure to evaluate its deviation
- how these "test pairs" are generated

is in the generic Evaluator class. See following sections for more details of these tasks.

- "3.3. Calculating a similarity measure's deviation from the gold standard" on page 15
- "4.3.1.2. Calculating a fitness value from a deviation" on page 22

The two classes, Evaluator<LocalSimilarityMeasure> and Evaluator<SymbolNode> actually do not exist. They are shown in the diagram to highlight that MeasureEvaluator actually extends Evaluator<LocalSimilarityMeasure> (and GP_SimMeasure_FF extends Evaluator<SymbolNode>, respectively). The reason that in the GP the evaluator and the fitness function is the same, is that no interpretation on the chromosome is needed. In GA the fitness function first interprets the chromosome and builds up a similarity measure, and than uses the MeasureEvaluator to evaluate it. The general evaluators are located in `simpack.learning`, the GP related evaluators are located in `simpack.learning.gp`.

## 8.3. Learning with the genetic algorithm

All the GA specific classes are located in `simpack.learning.ga`. For the genetic algorithm the open-source JGAP-Framework has been used. To improve the learning performance, an own NaturalSelector and an own GeneticOperator have been implemented according to the interfaces defined in the JGAP-Framework. This is done to have a selection strategy as presented in Section "4.3.2.4. Combination of the Strategies" (see p. 25) and a genetic operator as described in "4.3.3.3. Combination of the genetic operators" (see p. 27). This implementations can be found in the classes SimpackSelector and SimpackGeneticOperator respectively. To improve the runtime, the learning cycle has been implemented newly (outside the JGAP Framework). This is done in the class GA_Starter, where therefore a method exists to start an evolution.

Another important part of this packages are the so called GeneDataInterpretors. Their task is to transform a chromosome into a LocalSimilarityMeasure. They are used by the fitness function which is also part of the `simpack.learning.ga` package. A detailed instruction how to learn similarity measure using this package can be found in the appendix.

## 8.4. The Genetic Programming Framework

This framework has been developed to be able to use genetic programming to learn an amalgamation function. It is developed in an analogue way as the JGAP-Framework has been built. Also three interfaces, one for a fitness function, one for a genetic operator and one for a selector exist. Each of them is implemented for the case of learning an amalgamation function. What is not needed in the genetic programming approach, are the GeneDataInterpreters. Instead of them a so called GP_Executor has been developed. This is a class that is able to execute a genetic program, having its parse tree. For that, it maps the symbols, contained in the tree nodes on the defined Java-

methods which then will be invoked. Furthermore it maps the terminal symbols, which stand for a variable, to the actual value this variable has. These values can be set before executing the genetic program, and so the defined variables can be used as arguments (input values) for a genetic program. The GP_Executor is also the class that is able to produce an initial population of genetic programs to start the evolutionary process. Objects of the class GeneticProgram represents the actual programs (i.e. the individuals in the population). They consist of the parse tree (to execute them by the GP_Executor) and a fitness value, which will be set by the bulk fitness function (this is a function that calculates the fitness for each individual in the population).

## 8.5. Learning an amalgamation Function using GP

To learn an amalgamation function for a global similarity measure, the two classes GP_SimMeasureAnalyser and GP_SimMeasureFF are used.

First SimMeasureAnalyser is used to create an accurate GP_Executor. In this step, for each local similarity value, that the global measure would produce when its calculated, a variable will be defined in the GP_Executor. If the (GA-learned) amalgamation function should be exploited to learn the new one, all learned weights could also be added to the GP_Executor as constants[2]. The SimMeasureAnalyser also defines the functions that the genetic program may use, their symbols and the concrete Java-methods, which are associated with the function symbols.

This built up GP_Executor and the global similarity measure are given to the GP_SimMeasureFF, the fitness function for this task. When the fitness of an amalgamation function should be calculated, first all local similarity values will be calculated, using this global measure. These values will be set to the variables in the GP_Executor (SimMeasureAnalyser has defined these variables previously). Having done this, the GP_Executor is used to calculate the result which the learned amalgamation function produces. This then will be the global similarity value, which is used to calculate the deviation (as in the case of GA-learning). Note that always an amalgamation function for one single similarity measure is learned. This similarity measure can be a measure learned previously by the GA.

These classes are also located in the package `simpack.learning.gp`, together with the GP Framework's classes.

2 Technically they are variables, which will be set only once.

# 9. Conclusions & Future Work

In this chapter first a summary of the work done and presented in this diploma thesis is given. Afterwards we will see what benefits and drawbacks the applied methods come up with. In the last section proposals for future work will be presented.

## 9.1. Summary

At the very beginning of this diploma thesis we have introduced similarity measures and shown how the Local/Global principle works. Afterwards a possible Local/Global framework has been presented, which will be needed later on when similarity measures will be learned.

We have seen how the soundness of similarity measures can be evaluated using a gold standard and how such a gold standard can be created. Initially these two chapters had nothing to do with the real learning algorithm but this work has been done to have the basis for using machine learning of similarity measures.

Afterwards the details of the genetic algorithm (the first chosen learning technique) have been presented, always looking at the final goal of learning similarity measures. In the following evaluation part we have seen that the genetic algorithm comes up with very good similarity measures for the product data set. We have also seen that the approach works in principal also for the categorisation task of finding a property alignment between two ontologies, but that in absolute numbers, the results are are not as good as the results using the product data set. Therefore some proposals for future work are made at the end of this evaluation part. These proposals will be repeated together with further proposals for improvement in in the “Future work” section of this chapter.

Having all these things done, the method of genetic programming has been introduced and we have seen how we can exploit this technique to improve the (first by the GA learned) similarity measures. In the evaluation of this part, we have seen that genetic programming is able to find better amalgamation function than the (predefined) weighted average, and that genetic programming was able to improve the ontology alignment significantly, but results are still on a fairly poor level.

## 9.2. Benefits & Drawbacks

**Local/Global Principle**

The benefit of using the Local/Global principle is, that it can be used for nearly any kind of data structures. This is especially true with its recursive version, introduced in Chapter 2, where a local similarity measure can also be a global one, which has its own local similarity measures again. With this principle, complex- or aggregated objects, having every possible tree-like data structure, may be compared.

**Using artificial evolution**

The use of artificial evolution, which means the genetic algorithm and/or genetic programming, itself results in the following benefits.

- Less domain specific knowledge is needed to find a good similarity measure.
- It is a way to overcome the designer bias. The designer bias means, that if a human constructs something (e.g. a similarity measure), he always intuitively uses his conception of what the solution might be. This behaviour usually leads to the fact that not all possible solutions are considered.
- Using a gold standard is a good interface for transporting the domain knowledge of an expert into the learned similarity measure, without needing a domain expert, who is also a computer scientist.

**Genetic programming**

Genetic programming can learn an algorithm itself, unlike the GA which only learns a parameter setting for a defined algorithm. For instance an own amalgamation function, instead of just a parameter setting for a weighted average, can be learned. Therefore (compared to the GA, where the solution initially must be parametrized) with genetic programming even less knowledge about the characteristics of the solution is needed.

A drawback is, due to the closure property, GP is not able to learn a whole similarity measure to compare complex objects. The genetic algorithm and so the Local/Global framework is still needed.

Another drawback is that, GP compared to GA, needs larger populations and therefore much more runtime. The results of GP are also less predictable than those of using GA.

## 9.3. Future work

In this section we will see what future work can be done to improve particular results of the evaluations or to widen the scope of application of this framework.

**Future work concerning the Local/Global Framework**

As seen in Chapter 2 the Local/Global framework consists of four types of similarity measures and four types of configurations for the similarity measures. Including the similarity tree there would be five configuration types.

So more types of measures could be implemented (or built into the framework) to achieve that more accurate (global) similarity measures can be constructed. These could be measures that compare two data structures (e.g. tree edit distance[3]) or measures to compare long strings that have a textual character (e.g. TFIDF).

As seen in Chapter 2 the string and the number similarity weights consist of a "raw" similarity measure (distance measure in the case of number similarity) and a mapping that can be learned. The learning could be widened to the parameter setting of these "internal" measures.

Furthermore one might introduce a kind of a "similarity graph". Like a similarity tree, such a graph could be used to calculate a similarity table for symbol similarity measures. In the case of a similarity tree, a learned similarity of the nearest common parent node defines the similarity of two symbolic values. In a similarity graph the nearest path from one to another object could be calculated and learned values of each edge could be used to calculate a similarity value.

**Improving the results of the ontology alignment task**

The alignment task can be done with larger ontologies. In larger ontologies (at best with larger class hierarchies) the chance that two classes which are not semantically identical, but nevertheless have the same values for superclasses, subclasses and direct subclasses, will be lower. As mentioned in the explanation of the first experiment using the class hierarchy (see "Using property names, class names and class hierarchy of domain and range", on page 43), this is a reason that these values do

3 The tree edit distance measure implemented in SimPack had an unacceptable runtime for use it in a learning cycle. So the necessary first step before being able to use it would be an optimisation.

not really help finding the correct alignment. Therefore I predict better results using just a larger ontology.

More meta data of the properties can be taken into account. E.g. the cardinality could be exploited also. But this would have the same problem as the "position in hierarchy", the value of this attribute will be in most of the cases just 1. Therefore, for a lot of property pairs, which are not part of the correct alignment, the comparison of this attribute will lead to a maximum similarity.

The basic idea of using the class hierarchy was to identify the classes that represent each other semantically, and so to reason, that properties, where the range and the domain is the (semantically) same class, have a good chance to represent each other. Now the same procedure, as is used to find the alignment between the properties, can be used to find the class alignment. Having this once it can be exploited to find the property alignment.

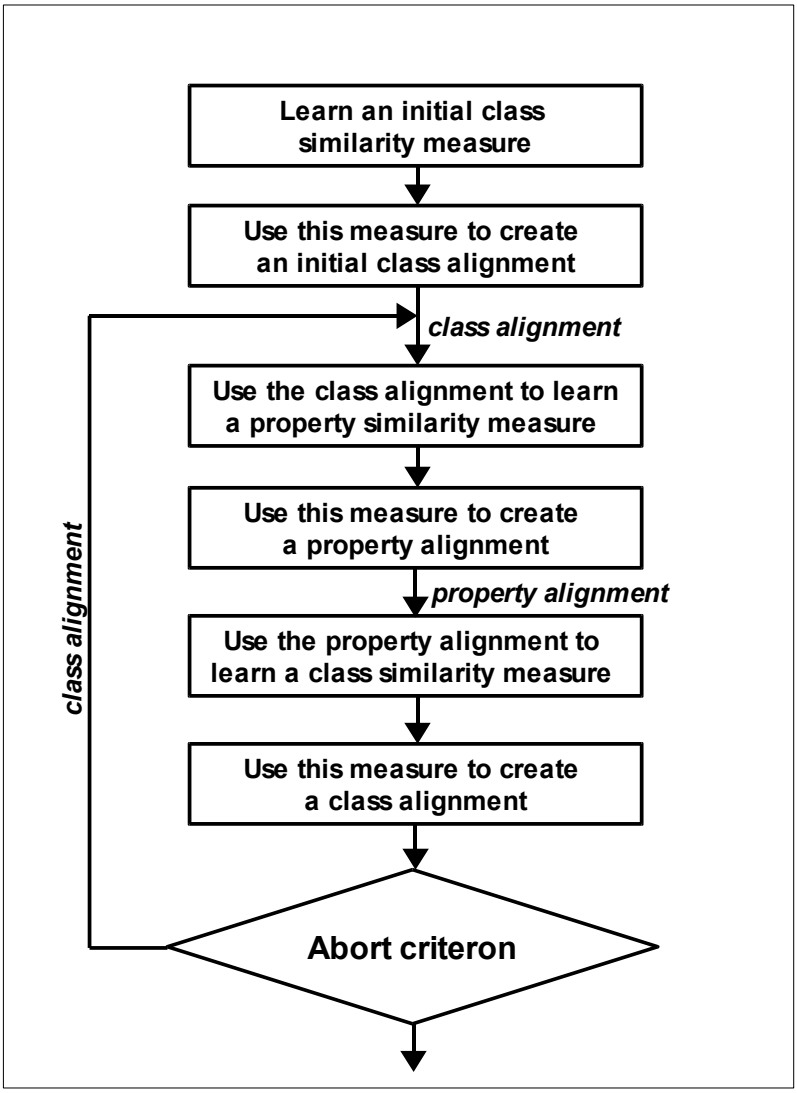

*Figure 47: Suggested learning cycle for an ontology alignment*

As already mentioned, the class alignment could be learned in a first step, analogue to the the property alignment. Looking at the (quite poor) results of finding the correct property alignment, there is also not much hope that finding the class alignment would perform much better. But to find the class alignment, the previously found property alignment can be exploited. So this could be done in a cycle. An initially quite insufficient class alignment will lead to a slightly better property alignment. Than this

property alignment again is used to learn a better class alignment and so forth. Figure 47 shows the imagined learning cycle. Note that in the Figure 47 the class similarity measure is learned initially and therefore an initial class-alignment will be created. This is arbitrarily defined, one can also start the learning cycle with an initial property alignment.

**Doing more evaluations**

In this diploma thesis evaluations with two datasets have been done, the product dataset and the ontology alignment task. Of course more different data sets can be used to do more evaluations of the whole framework. As the Local/Global principle, especially in its recursive version, it is very universal. Objects of almost every thinkable data structure can be compared with each other by an accurately configured global similarity measure.

**Refactoring SimPack**

Using the SimPack library was fairly complicated because a lot of different return- and argument types are used. This may be the result of the different developers working on the SimPack and their individual tastes. I would propose to use primitive types, and instead of collection, sets, vectors and all these classes, to use simple arrays of the particular objects, wherever possible. This simply would make it easier understandable for a user (e.g. when he reads the Java-doc) and, if less different classes for argument and return types are used, a lot of transformation work could be saved. Another benefit (especially of using primitive types) would be that, simpler data structures will usually lead to less memory- and runtime[4] consumption.

So we can see that a lot of further improvement is possible. These further improvements will make learning similarity measures more efficient, more universal and easier to apply.

4 I reimplemented the Levensthein algorithm and got a 4 times better runtime just by using a 2-dimensional array of a primitive type instead of a matrix object.

# 10. References

[Stahl 2004] A. Stahl. *Learning of Knowledge-Intensive Similarity-Measures in Case-Based Reasoning*. PhD thesis, Chapter 3, University of Kaiserslautern, Germany, 2004. Verlag dissertation.de, Band 986.

[Gabel 2005] T. Gabel. *On the Use of Vocabulary Knowledge for Learning Similarity Measures*. In Wissensmanagement, pp. 253-258, 2005.

[Dawkins 1986] R. Dawkins. *The blind watchmaker*. 1986. Penguin Books Ltd.

[Stahl and Gabel 2003] A. Stahl and T. Gabel. *Using Evolution Programs to Learn Local Similarity Measures*. In Proceedings of the 5th International Conference on Case-Based Reasoning (ICCBR 2003), Trondheim, Norway, June 2003.

[Pfeifer and Scheier 1999] R. Pfeifer and Christian Scheier. *Understanding Intelligence*. Chapter 8 p.275. 1999. MIT Press.

[Euzenat *et al.* 2004] J. Euzenat, Y. Sure, O. Corcho, H. Stuckenschmidt, L. Palmer, N. Noy and D. Loup. *EON Ontology Alignment Contest* `http://oaei.ontologymatching.org/2004/Contest/` 2004.

[Koza 1992] J. Koza. *Genetic Programming*. 1992, MIT Press.

[Bernstein *et al.* 2005] A. Bernstein, E. Kauffmann, C. Kiefer and C. Bürki. *SimPack: A Generic Java Library for Similarity Measures in Ontologies*. Technical report, Department of Informatics University of Zurich. 2005.

[Bernstein and Kiefer 2006] A. Bernstein and C. Kiefer. *SimPack Project Page* `http:://www.ifi.unizh.ch/ddis/simpack.html` 2006.

# Appendix

## Appendix A - Using the Local/Global Framework

This framework (for local similarity measures) can be used in two ways:

1. It can be used just to define concrete similarity measures.
2. To learn meaningful configurations for similarity measures, which was the point in this diploma thesis.

### A.1. To configure Similarity Measures

If one wants to use this framework just for creating (known) similarity measures, one can just instantiate the correct configuration classes, set the parameter values, in a way he thinks is meaningful, and create the similarity measures. There are two ways to get from the configurations to the working measure. Either the LocalSimilarityMeasure is instantiated and it's desired configurations is given to the constructor or, the more elegant way, one can call the getMeasure() method of the configuration object. The getMeasure() method is defined in the Interface ILocalSimilarityConfiguration so that it is ensured that every configuration class has to implement it.

### A.2. To learn Similarity Measures

To learn a meaningful configuration, it is necessary to learn all parameter values. If a genetic algorithm is used for this, the number of parameters defines the length of the chromosome (or the number of genes). If one just wants to learn for example one similarity table, he can easily calculate how many free parameters need to be set, and so start a genetic algorithm. In each fitness test he would interpret the chromosome and build up the similarity table. But calculating the length of the chromosome needed and interpreting it, will get more and more difficult, if one wants to learn complexer measures (such as recursive object measures). If something changes (e.g. one wants to ensure a similarity table to be symmetric) the interpreter for the chromosome has to be reprogrammed.

To avoid this work, another type of helpful classes has been created, the so-called GeneDataInterpreters. These classes build the bridge from the genetic algorithm (which knows nothing about similarity) to the configurations and measures (which know nothing about the genetic algorithm). Basically an instance of a GeneDataInterpreter (again 4 types exist, analogue to the configurations, and they also can be arranged recursively) is able to transform a number of genes into a configuration. Furthermore it can answer the question how many genes are needed for it's configuration and can create a sample chromosome. These two features are needed to calculate the length of the needed chromosome automatically. Once having a configuration, it is easy to create a similarity measure from it, which can be evaluated (e.g. to calculate it's fitness).

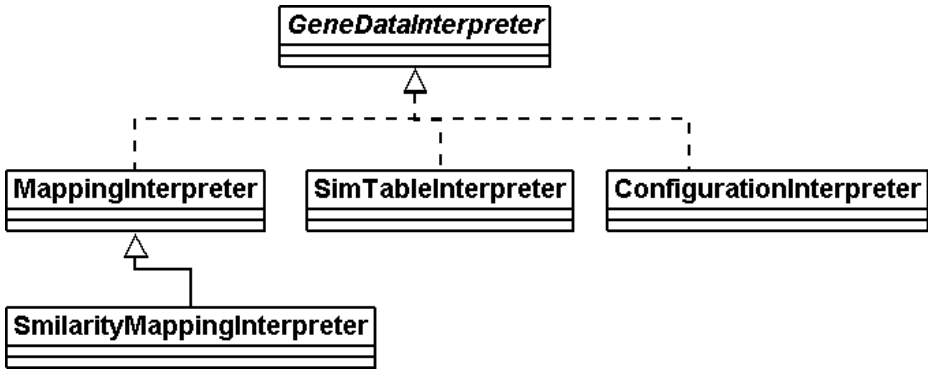

*Figure 48: Class diagram of the different GeneDataInterpreters*

## A.3. An Example how to use this Framework

This section should give a concrete example, how to configure a global similarity measure and how to use the GeneDataInterpretors to learn a similarity measure.

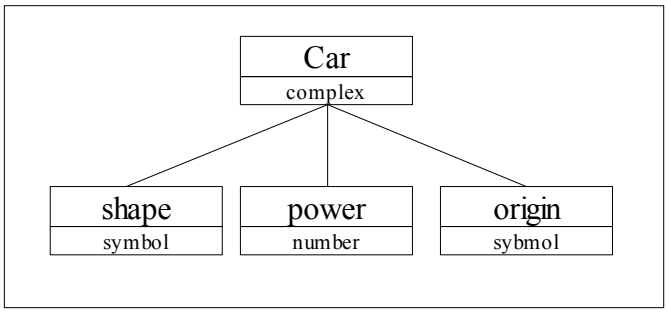

*Figure 49: The structure of the car objects used in the example*

**Configuring a similarity measure**

Assume that a simple global similarity measure to compare cars should be configured. Three attributes of a car should be used: shape, power (in kW) and origin (the country where it was produced). To start, we instantiate the ObjectSimlarityConfiguration for our desired global similarity measure.

```
ObjectSimilarityConfiguration osc = new ObjectSimilarityConfiguration();
```

Having this done, the configurations for the local similarity measures can be set up. For the 2 symbolic attributes we need a SimilarityTable, for the numeric one, a Dist2SimMapping. Additionally for shape we wants to use a taxonomy to calculate the similarity table (see Fig. 50).

```
Symbol[] shape = new Symbol[4];
shape[0] = new Symbol("Cabriolet");
shape[1] = new Symbol("Coupé");
shape[2] = new Symbol("Stationwagon");
shape[3] = new Symbol("Limousine");
SimilarityTable tableShape = new SimilarityTable(shape);
```

"shape" defines the symbols, which this table may compare, this array hast to contain all possible values the attribute may have. Afterwards, build up the SimilarityTree and pass it to the similarity table:

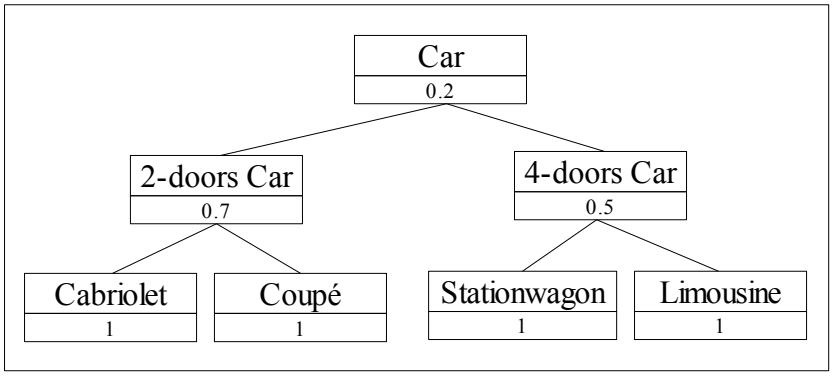


*Figure 50: Similarity Tree for the attribute shape*

```
String[] twoDoors = {"Cabriolet", "Coupé"}
String[] fourDoors = {"Limousine", "Stationwagon"};
SimilarityTree[] cars = new SimilarityTree[2];
cars[0] = new SimilarityTree("2-door", 0.7, twoDoors);
cars[1] = new SimilarityTree("4-door", 0.5, fourDoors);
root = new SimilarityTree("Car", 0.2, cars);
tableShape.setSimilaritiesByTree(root);
```

To add this configuration to the global configuration use this:

```
osc.addAttribute("shape", 0.4, tableShape);
```

The first attribute of the function represents the key of the property, which contains the values to compare in the car objects, the second is the weight of this property.

As next, define the configuration for the numeric similarity measure used to compare the power of the cars. For this a Dist2SimMapping is used. Using our common sense (vocabulary knowledge) we want to have a measure that is symmetric.

```
double[] powerValues = { 1, 0.8, 0.5, 0.25, 0.1, 0 };
Dist2SimMapping powerMapping = new Dist2SimMapping(0, 100,
          Dist2SimMapping.DIFFERENCE, powerValues);
powerMapping.setSymetric(true);
osc.addAttribute("power", 0.3, powerMapping);
```

The similarity values in the array "powerValues" will be uniformly distributed over the difference range from 0 to 100. Finally the mapping is added to the global configuration object.
For the third attribute another similarity table is needed. Unlike for the attribute "shape" a similarity table could also be manually defined like the distance-to-similarity mapping, but a 2 dimensional array will be needed.

```
double[][] simValue = { { 1, 0, 0.4, 0, 0.3 }, { 0, 1, 0, 0.75, 0.3 },
          { 0.4, 0, 1, 0, 0 }, { 0, 0.75, 0, 1, 0 }, { 0.3, 0.3, 0, 0, 1 } };
Symbol[] country = new Symbol[5];
country[0] = new Symbol("Germany");
country[1] = new Symbol("France");
country[2] = new Symbol("Sweden");
country[3] = new Symbol("USA");
country[4] = new Symbol("Japan");
SimilarityTable tableOrigin = new SimilarityTable(country, simValue);
osc.addAttribute("origin", 0.3, tableOrigin);
```

Having all this done, the measure can be created:

```
LocalSimilarityMeasure carSimMeasure = osc.getMeasure();
```

This measure will be able to compare objects of the class Hashtable<String, Object> whereby they should have the keys "shape", "power" and "origin" for the attribute values.

**Create a GeneDataInterpreter**

To learn a similarity measure using the genetic algorithm, a so called GeneDataInterpreter is needed.

In this section it will be shown how to build such an interpreter for the car example. First instantiate the interpreter for the global measure.

```
ConfigurationInterpreter carInterpreter = new ConfigurationInterpreter();
```

Having this, create the interpreters for the local measures and add them to the global one:

```
SimTableInterpreter shapeInt = new SimTableInterpreter(shape);
shapeInt.setTreeStructure(root);
carInterpreter.addSubInterpreter("shape", shapeInt);
```

Note that "shape" should represent the previously introduced symbol array, and "root" stands for the similarity tree defined in the previous section. The similarity values, which have been set there, will not be used by the interpreter. To set these, the genes of the chromosome to be interpreted, will be used.

For the similarity mapping the following interpreter can be defined:

```
MappingInterpreter powerInt = new MappingInterpreter(0, 100, 6,
          Dist2SimMapping.DIFFERENCE);
powerInt.setMonotonic(true, Dist2SimMapping.DECREASING);
powerInt.setSymetric(true);
carInterpreter.addSubInterpreter("power", powerInt);
```

The third argument of the constructor indicates how many values may be learned to create the mapping.

For the last symbolic attribute "origin" the following interpreter can be used:

```
SimTableInterpreter originInt = new SimTableInterpreter(country);
originInt.setSymmetric(true);
carInterpreter.addSubInterpreter("origin", originInt);
```

Note that "country" stands for the symbol array that was created in the last section.

At this point this interpretor can be used to learn a concrete similarity measure that compares the car objects in our example. Additionally the interpreter can answer the question how many genes are needed to encode all parameters and he can create a sample chromosome (technically an array of genes) for the JGAP-Framework.

# Appendix B - GA Learning

To learn a similarity measure, first an evaluator is needed. To instantiate this a training set and a gold standard is needed. Another option is to have a target measure that calculates the similarity values for the gold standard.

So first somehow the training set has to be loaded (form a database, csv-file, ontology or whatever the data source might be[5]) and brought in a form that it is an array of Hashtable<String, Object>. The gold standard will be an 2-dimensional array of doubles, indicating the desired similarity between two objects, whereby *goldStd*[*i*][*j*] should be *desiredSimilarity(trainingSet[i], trainingSet[j])*. The MeasureEvaluator than can be instantiated like this:

```
MeasureEvaluator me = new MeasureEvaluatorGoldStandard(trainingSet, goldStd);
```

or

```
MeasureEvaluator me = new MeasureEvaluatorTargetMeasure(trainingSet,
          targetMeasure);
```

Having this done, the fitness function can bin instantiated using the evaluator and the GeneDataInterpreter introduced Appendix A.3. Once having the fitness function, out of it the so called BulkFitnessFunction can be created, which calculates the fitness of each individual in the population at once.

```
SimMeasureFitnessFunction ff = new SimMeasureFitnessFunction(me,
          carInterpreter);
SimMeasureBulkFitness smbf = new SimMeasureBulkFitness(ff);
```

These two steps are already implemented in the GA_Starter, the class which starts and controls the learning cycle.

5 Some classes to do this are in the package `simpack.data_access`

To start a learning cycle, at least a GeneDataInterpreter and an Evaluator is needed. Optionally a specific GeneticOperator and/or a specific NaturalSelector can be used. Otherwise the default SimpackGeneticOperator and the default SimpackSelector will be used.

```
populationSize = 50;
generations = 250;
GA_Starter.evolvePopulation(me, carInterpreter, populationSize, generations);
```

This method returns an array of Chromosomes. From every generation the chromosome of the best performing individual is in this array. “carInterpreter” is the reference to the GeneDataInterpreter used.

**Configuring the GeneticOperator and the NaturalSelector**

The SimpackSelector can be configured for example as follows:

```
int eliteSize = 10;
int grr  =  3;
int jokers = 25
INaturalSelector selector = new SimpackSelector(eliteSize, grr, jokers);
```

Where eliteSize defines the size of the elite (in percent of the whole population), which has a guarantee to be selected grr times (grr = guaranteed reproduction rate). “jokers” defines how many percent of the individuals will be selected regardless of their fitness.

The SimpackGeneticOperator can be customized like this:

```
double crossoverRate = 0.75;
double mutationRate = 0.4;
double mmi = 0.2;
GeneticOperator go = new SimpackGeneticOperator(mutationRate, crossoverRate,
          mmi);
```

Where mutationRate and crossoverRate define the chance for each individual do reproduce by crossover or mutation, mmi defines the maximum mutation impact.

# Appendix C - GP Learning

To learn with genetic programming, first also an evaluator is needed. This evaluator is in this case identical with the fitness function, because no GeneDataInterpreter is needed in GP. Analogue to GA-leaning, to instantiate the evaluator, a training set and a gold standard (optionally a target measure) is needed. But furthermore the evaluator for an amalgamation function needs to know the ObjectSimilarityMeasure, for which the amalgamation function should be learned, and a GP_Executor, which is used to execute the genetic programs. The measure can be a previously GA-learned similarity measure called "measure" in the following code snippet. Having this measure, also the GP_Executor can be created using the class SimMeasureAnalyser :

```
GP_Executor<Double> gpEx =  GP_SimMeasureAnalyser.createGPExecutor(measure);
GP_SimMeasureFF ffTraining = new GP_SimMeasureFF_GoldStandard(gpEx, measure,
          trainingSet, goldStd);
```

The selectors and the genetic operators work similar to the ones used in GA, the genetic operator has only one parameter, crossoverRate, because no explicit mutation is implemented. Also there exists no method that starts and controls the learning cycle. The learning cycle could be set up like this:

```
int gpGenerations = 50;
int gpPopSize = 500;
GP_CrossOverOperator go = new GP_CrossOverOperator();
GP_SimpackSelector sel = new GP_SimpackSelector();
GeneticProgram[] pop = gpEx.createInitialPopulation(gpPopSize);
GP_BulkFitnessFunction bff = new GP_BulkFitnessFunction(ffTraining);

for (int n = 0; n < gpGenerations; n++) {
          bff.evalute(pop);
          GeneticProgram[] selected = sel.select(gpPopSize, pop);
          pop = go.operate(gpPopSize, selected);
}
```

Note that in this GP-Framework the population is represented simply by the array of individuals the population contains.