The Evolution of LIMES - Learning a mapping using genetic programming

Klaus Lyko
University of Leipzig

September 21, 2011

Abstract

only for a paper version Linked Data, Object Matching, learing using GP

Contents

1 Introduction

still left

2 Related work

As of now:

- instance matching between data sources, without prior knowledge about schema and ontologies **Araujo et al.** [3]
- **SILK** as the state of the art approach [5, 4]
- **LIMES** Paper - maybe ISWC2001 submitted ????

3 Overview

3.1 Genetic programming

3.1.1 basic literature

Traditional evolutionary algorithms are applied to optimization problems. GP is part of the machine learning branch. Kapitel 6 'Genetic Programming'.[1]

> Besides the particular representation (using trees as chromosomes) it differs from other EA strands in its application area. While EAs discussed so far are typically applied to optimisation problems. GP could instead be positioned in machine learning.

Representing individuals: 'defining the syntax of the trees [...] This is commonly done by defining a **function set** and a **terminal set**.' Elements of the function set are internal nodes while elements of the Terminal set are allowed to be leaves. For each symbol in the function set the arity (number of attributes) must be given. Further 'for the complete specification of thge syntax a definition of correct expressions (thus trees) based on the function and terminal set must be given'[1]
Kapitel 5 'Overview of Genetic Programming' S.73-78. Bzw. Kapitel 6 'Detailed Description of Genetic Programming'S.79-120. In Introduction why GA not suitable for many problems: fixed length representation etc.[2]
Section 9 about GA. 9.5 short overview of GP. With referencing Kozas research.

3.2 The Limes Framework

TODO: Descibe the LIMES framework.

3.3 The genetic programming library JGAP for Java

TODO: Descibe the JGAP library.

4 Implementation

In order to profide a learning mechanism for LIMES using genetic programming (GP) we have to combine the evolution capabilities of the JGAP library with the mapping work done within the LIMES framework.

4.1 Basic approach

As we will use Genetic Programming to implement our learner, we will use the Genetic Programming facilities of the JGAP library. To evolve the mapping strategies used by LIMES to take care of the actual matching process, we will need to implement at least:

- A class extending *org.jgap.gp.GPProblem* - currently ExpressionProblem - Defining the basics of our GP (Genes (Functions, Terminals, Fitness-Function etc.). As such describing the problem to be solved by genetic programming.
- Classes for our **Commands** - all decending from *org.jgap.gp.CommandGene*
- As such also Terminals used in the evolution.
- A **FitnessFunction** extending *org.jgap.gp.GPFitnessFunction*, as we can only define any fitness of a solution by running the LIMES pipeline, the class have to controll this process.
- We may need **additional data** for the evolution process. JGAP profides the interface *org.jgap.IApllicationData*for this purpose. We can attach a class implementing it to our chromosomes, this data isn't touched by JGAP. Here we specify basic configuration settings. Such as source and target informations.
- FUTURE Our own genetic **operators** can be specified by implementing *org.jgap.gp.IGPGeneticOperator* - see *examples.gp* for some implementations.
- FUTURE A genetic program can interpreted as a tree, where Commands are the nodes. We may need to **validate nodes** in our tree during the evolution process. This is done by *org.jgap.gp.INodeValidator*.

4.2 Problem

To define among other things which functions and terminals are used to build a program to solve the problem we defined the *ExpressionProblem* class extending GPProblem. The main purpose is to create a Genotype of the individuals. A genotype is a predefined list of chromosomes and sets of Functions, Terminal and Variables used to build the program trees of the chromosomes. JGAP will call the *create()* method to build the genotype.
We have (as of now?) individuals with two chromosomes: The first is a program

tree building the metric expression used by LIMES to map the instance of both knowledge bases. The second chromosome only accepts one Terminal returning a double and is used to evolve the global acceptance threshold for our metric expression.

4.3 Configuration

A *GPConfiguration* class controls central steps of the evolutionary process. Among others population size, probabillity of mutations and the used fitness function and the fitness evaluator. As we need certain informations especially the KBInfo instances of the source and target knowledge bases throughout the very stages of the evoltionary process (defining the genotype, evaluating individual soltutions) and the GPConfiguration is accessible anywhere we have defined the *ExpressionConfiguration* class extending *GPConfiguration*.

4.4 Functions and Terminals

We basically want to evolve expressions to decide which combination of source and target properties is best to indentify the same objects of both knowledge bases. So a genetic program is basically representing a metric expression used by LIMES to compare instances of the two knowledge bases. Therefor an evolved program is a combination of supported operators, working on certain properties of the knowledge bases. Such a program can be interpreted as a tree. Where nodes a certain commands, with a specified number and type of parameters, returning a specific type.

4.4.1 Properties as Terminals

The leafs of our programs are in the most cases properties of the source or target knowledge bases. As such they are *Strings*. As we have a specified list of properties, we don't change a property in the evolution process. We may swap it with another property though. Therefore we implement the properties as non-changeable Terminals. Each knowledge base has specified property list as defined in a LIMES configuration file. So instead of directly defining Terminal for those Strings we use the indices of the properties in the knowledge base property list as nodes in our gp programs.
To avoid useless metric expressions working on properties of the same knowledge base (e.g. "trigram(x.title, x.title)") we define two Terminals with different sub return types (defined in the nested enumerater *ResourceTerminalType* class in the ExpressionMain class). Those terminals will return a integer value representing the index of the property in the property list of the source or target knowledge base.

4.4.2 Basic functions

We now describe the functions used to build metric expressions in the evolutionary process.

Atomic Measures The basis for all metric expressions (working on properties of type String) in LIMES are implementing the *de.uni_leipzig.simba.measures.string.StringMeasure* interface. Those atomic measures are Cosine, Jaccard, Levensthein, OverlapMeasure and Trigram. All comparing the similarity of two String values. The central method is *getSimilarity()* returning a double value. This value is later compared to given similarity thresholds. All atomic expression have the same structure: *measure(x.prop1, y.prop2)*. Whereas measure is the name of the sprecific atomic measure, e.g. "trigrams". To define a JGAP command for these atomic measures we defined the *SimilarityCommand* class extending CommandGene and implementing the interfaces *IMutateable* and *ICloneable*[1].
A SimilarityCommand in a CommandGene expecting three parameters (child nodes in the program tree) and per default returning a String.class value thus results in a call to the execute_object() method while executing this command. We expect the first two childs (or parameters of this function) to deliver also values of String.class but also to have a specific sub return type. Thus making sure, that a SimilarityCommand node only accepts Terminals representing source properties as first child and Terminals representing properties of the target knowledge base as the second one. The third child is also specified by a sub return type and must be a double terminal. Is the SimilarityCommand the root node of the program tree, the individual will represent a atomic measure (e.g. "trigrams(x.prop1, y.prop1)") so a third parameter is unnecessary. But as a atomic measure could be part of more complex metric expressions, particularly boolean measures such as "AND(sim(x.prop1, y.prop2)—threshold1, sim(x.prop2, y.prop2)—threshold2)" we would also have to evolve thresholds as part of this SimilarityCommands. The LIMES Parser for metric expressions ignores those appended thresholds ("—threshold"). So as of now we just add this thresholds to each SimilarityCommand, but only in more complex measures such as boolean expressions like AND and OR these additional thresholds are actually considered by the LIMES framework. TODO We might define two(3? for MULT and ADD?) SimilarityCommands for cleaner expressions. Is there a way while executing programs to differentiate if the SimilarityCommand is a atomic measure or part of a complex command? We could use different execute methods. But a cleaner implementation could be by defining different ones.

Property Mapping As of now we support a genetic evolution of instance matching based on a prior done property mapping. It is not possible to consider the restrictions by a prior done proptery mapping at the start of the evoultionary process. As we would have to dynamically define multiple SimilarityCommands expecting different child nodes. For each property mapping we would have to add one SimilarityCommand expecting 2 certain child nodes with special sub return types to the evolution configuration. As this is no practical way we rather check if a given metric expression fulfils the restrictions of the property mapping during the evolutionary process. For each individual (a GP program)

[1] both defined in the org.jgap.gp package.

i check before calculating the fitness value whether or not it is comparing two properties, which are not part of the property mapping done before. If so we will give this individual a bad fitness value, before even calculating any matches. There saving needless comparing resources.
TODO Perhaps there is a more elegant way. It seems, that Node validation could be used, but i'm not sure.

Boolean Measures As of this stage we only support the following boolean measures to build complex metric expressions:

- AND - expecting two similarity measures over properties with thresholds (e.g. "AND(trigrams(x.title, y.title)—0.9, cosine(x.authors, y.authors)—0.8)"). The result will be a mapping holding all pairs of instances satisfying both similarity measures. LIMES constructs mappings between the two knowledge bases using the atomic measures and filtering with the given thresholds. Then both mappings will be merged only accepting the overlap.
- OR - also expecting two similarity measures with thresholds. The result will hold all instances which satisfy either Similarity measure.
- XOR - as above but using the exclusive or to combine both mappings.

As of now all boolean measures are implemented in seperate classes, but an anlogue implementation using on class and defining the actual command via the constructor is possible.

4.5 Fitness

We have to develop a FitnessFunction extending *GPFitnessFunction*. This class will have to run the LIMES pipeline in order to decide how fit our solution is. We won't use the JGAP Genetic Programming Facilities completly for this process. We will rather build the expression out of the first chromosome, then use them to run the LIMES pipeline. We defined the class *ExpressionFitnessFunction* for this purpose. Extending *GPFitnessFunction*. On evaluating an individual the *evaluate*() method will be called.
On constructing a instance of this class, we initiate and fill the Caches for the source and target knowledge base. The class also holds an instance of a Mapper (right now a *SetConstraintsMapper* out of the LIMES *mapper* package). This mapper will later perform the actual mapping process. For this we need the metric expression. The GPProgram parameter of the evaluate function consists of the defined number of chromosomes (defined by the genotype of the problem in our *ExpressionProblem* class). The program tree building the actual metric expression is the first chromosome with index 0. Thus we call the *execute_object()* method of the first (root) node to build expression string. The global acceptance threshold is defined by the second chromosome, whereas only one command (a Terminal returning a double value) is part of the set of allowed

commands (see section problem definition for details).
To get the resulting mapping we call the *getLinks()* method with the metric expression out of chromosome 0 and the threshold of chromosome 1.

References

[1] Smith J.E. Eiben, A.E. *Introduction to Evolutionary Computing*. Springer, Berlin Heidelberg, 2007.

[2] John R. Koza. *Genetic Programming*. MIT Press, Cambridge MA, 1992.

[3] Araujo S., Hidders J., Schwabe D., and Vries A. P. SERIMI - Resource Description Similarity, RDF Instance Matching and Interlinking. In *The 10th International Semantic Web Conference (ISWC 2011)*. Bonn, Germany, 2011. submitted.

[4] J. Volz, C. Bizer, M. Gaedke, and G. Kobilarov. SILK - A Link Discovery Framework for the Web of Data. In *In Proceedings of Linked Data on the Web (LODW2009) workshop at WWW2009*, Madrid, Spain, 2009.

[5] Julius Volz, Christian Bizer, Martin Gaedke, and Georgi Kobilarov. Discovering and Maintaining Links on the Web of Data. In Abraham Bernstein, David Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009*, volume 5823 of *Lecture Notes in Computer Science*, pages 650–665. Springer Berlin / Heidelberg, 2009.