Univerzita Karlova v Praze, Filozofická fakulta
Katedra logiky

Jan Frauknecht

Koza a Prolog
Koza and Prolog
Bakalářská práce

Vedoucí práce: Mgr. Petr Švarný

2014

Děkuji Petrovi za odvahu a posléze za ocelové nervy.
Děkuji své rodině za nikdy nekončící podporu.
A v neposlední řadě děkuji Janě za to, že mě v nejvyšší nouzi hnala do hor!

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a že jsem uvedl všechny použité prameny a literaturu.

V Praze 2. ledna 2014

Jan Frauknecht

Abstract

Tato práce uvádí vztah umělé inteligence ke genetickému programování a některé vlastnisti logického programování. Hlavním cílem práce ovšem je naprogramovat algoritmus genetrického programování. Tento program operuje s logickými programy. Algoritmus je implementován v SWI-Prologu. Práce obsahuje popis zdrojového kódu této implementace a výsledky jejího testování. Testování implementace nabízí několik možností budoucího rozšíření práce.

Abstract

This paper introduces the artificial intelligence background of genetic programming and some properties of logical programming paradigm. However, the main task of this work is to create the genetic programming algorithm that operates with the logical programming paradigm. SWI-Prolog is used for the actual implementation of such a program. This implementation is in detail described. The testing of this implementation shows some possible path for the future work.

Contents

List of used acronyms **6**

1 Prologue **7**

2 Biologically inspired algorithms **8**

2.1 Evolutionary algorithms . . . 8
2.2 Genetic algorithm . . . 9
2.2.1 Operators of reproduction . . . 10
2.2.2 Operators of selection . . . 11
2.3 Genetic programming . . . 12
2.4 Lamarckism . . . 13
2.5 PS2O . . . 14
2.6 Bee colony algorithm . . . 15

3 Logic and EA **17**

3.1 Logical programming . . . 17
3.2 Prolog . . . 18
3.3 Motivation . . . 18

4 PROGP **20**

4.1 Properties of LP in context of GP . . . 20
4.1.1 Structure of the program . . . 20
4.1.2 Variables . . . 21
4.2 PROGP vs. PROGA . . . 21
4.2.1 Alteration of PROGA . . . 22
4.3 PROGP description . . . 22

5 Source code of PROGP **24**

5.1 General idea of PROGP . . . 24
5.2 ga/0 . . . 25
5.2.1 initialise/12 . . . 25
5.3 runExistingPop/1 . . . 26
5.3.1 getPars/11 . . . 26
5.4 run/14 . . . 27
5.4.1 reproduce/12 . . . 28
5.4.2 xovers/10 . . . 28
5.4.3 mutation/10 . . . 30
5.4.4 lamarckian_evolution/7 . . . 31
5.5 fitness/3 . . . 32
5.5.1 shapeError/2 . . . 32

5.5.2 curGraph2pl/2 32
5.5.3 loadInputs/4 32
5.5.4 runError/6 33
5.5.5 taxManager/5 33

6 Run of PROGP **34**
6.1 First member 34
6.2 Member 36
6.2.1 Genome evolution example 37
6.3 Concatenate 41
6.3.1 Experimental settings searching 41
6.3.2 Concatenate run 43

7 Conclusion **44**
7.1 Size of search space 44
7.2 Why is PROGP unsuccessful? 46
7.3 PS2O application 47
7.4 Prolog recursion handling 47
7.5 Prolog failure and signal errors 48
7.6 Epilogue 48

A List of attachments **49**

B Translation algorithm **50**

C Description of the attached files content **51**

References **52**

List of used acronyms

PROGP PRolog Genetic Programming

PROGA PRolog Genetic Algorithm

BI Biologicaly Inspired

GA Genetic Algorithm

GP Genetic Programming

AI Artificial Inteligence

EA Evolutionary Algorithm

LP Logical Programming

I/O Input/Output

SLD Selective Linear Definite

1 Prologue

This headline is the first, but not the last occurrence of a word that is pronounced `prolog`. This particular occurrence is very specific because it refers to the first chapter that introduces the following text. Every other occurrence refers to the programming language.

The name of the work should evoke a connection to genetic programming. John Koza is the author of the idea of genetic programming [Koza 1992]. The first section of this text introduces some examples of biologically inspired algorithms. The rest of the text focuses mostly on genetic programming itself.

The goal of this thesis was to create a genetic programming implementation in Prolog (the result is called PROGP). Prolog is a programming language which was first introduced in the 1970's. Prolog was supposed to be a breakthrough in artificial intelligence. However, the interest of programmers has gradually declined due to its inefficiency. Nowadays Prolog is a tool mostly for logicians [Bratko 2001].

There has been an attempt to implement genetic programming for logical programming [Tang et. al. 1998]. It was designed in the programming language Lisp and the result was compared with two other artificial intelligence approaches. The genetic programming approach was significantly less successful in the comparison. PROGP uses some suggestions to create better fitness function from [Tang et. al. 1998].

The description and testing of PROGP are covered in the third and fourth sections. The resulting program is an alteration of Olsson's PROGA [Olsson 1996]. PROGA is a genetic algorithm implementation in SICSTUS Prolog. The translation of PROGA into SWI-Prolog is part of the goal. PROGP is also implemented in SWI-Prolog.

PROGP testing shows limits of the genetic programming approach. PROGP was tested on three different tasks. Only the most basic one was successful. PROGP also shows some limits to the SWI-Prolog distribution. Some not documented Prolog errors occurred during the programming and testing of PROGP.

2 Biologically inspired algorithms

Biology came with many general concepts that can be used as artificial intelligence (AI) algorithms. These algorithms are called biologically inspired (BI) algorithms. There are three large groups of BI algorithms. These are evolutionary algorithms, ecologically inspired algorithms, and artificial swarm intelligence algorithms [Binitha et. al. 2012].

There are four algorithms described in this section. These algorithms are taken as examples and most of them are mentioned later on. All three groups of BI algorithms are represented. First there are the evolutionary algorithms (EA), whose examples follow the historical development of the idea. First there was a general idea of using the evolution principle as an AI algorithm. The genetic algorithm has been the first application of this principle. Genetic programming is one of the latest results of EA.

The remaining two groups are represented only by one example. Ecological algorithms are introduced by the PS2O algorithm. It is possible to apply the PS2O algorithm as an extension of genetic programming in logical programming (i.e. PROGP) that will be introduced later. The last group of BI algorithms includes swarm intelligence algorithms and it is represented by the simulated bee colony algorithm.

All examples and algorithm definitions are taken from [Binitha et. al. 2012].

2.1 Evolutionary algorithms

The oldest and the best known BI algorithm is the evolutionary algorithm. Evolution refers to Charles Darwin's theory of evolution by natural selection. This theory was the inspiration of 1950's programmers like Holland, Ross, Bremermann, and Friedberg [Back et. al. 1997].

For purposes of this work it is necessary to distinguish the genome and the individual[1]. For the EA, the genome is the individual representation stored in the computer memory (and it is this genome that is evolved). Fitness can be counted only for an individual (not for the individual genetic code). This distinction was not introduced by Charles Darwin. The discovery of genotype was made in the 20th century[2].

[1]Biology distinguishes the difference of phenotype and genotype. Evolution runs on the level of genotype (the genetic code). Phenotype is the common name for every actual expression of genome [Dawkins 2006].

[2]Heredity. (2014). In Encyclopaedia Britannica. Retrieved from
http://www.britannica.com/EBchecked/topic/262934/heredity
[Britannica 2014]

The simulation of evolution is provided by an EA. The individuals usually represent certain solutions of a problem. Although it is possible to think of a self-reproducing program[3], EA has usually an overlord[4].

The general principle of an evolutionary algorithm is:

1. Make randomly the first generation of genomes.
2. Evaluate the whole population by a fitness function.
3. Check whether the goal was reached. If so, end with success. If not, continue.
4. Make the next generation:
   (a) use an operator of reproduction on the population.
   (b) use an operator of selection on the population.
5. Go to point 2.

This is a general idea common to all EA. The overlord decides how often the operators are used and it creates copies of the individuals.

The details in this paragraph are usually used but are not necessary. The size of a population is fixed. The operators of reproduction are crossover and mutation. The operator of selection is tournament. All these operators are described in the following section.

The various EA differ mostly by their representation of the genome.

2.2 Genetic algorithm

The most basic (and the oldest) evolutionary algorithm is called genetic algorithm (GA). The genome is in this case a binary vector of a certain length. This binary vector can represent almost anything you can imagine, for example the bag in the famous knapsack problem[5].

[3]Such program creates a reproductions (copies) of itself. It should create copies with some mistake or error (see section 2.2.1) to simulate evolution. Such program would be in fact a new replicator as introduced in [Dawkins 2006]. It should start a new epoch of evolution.

[4]The overlord is a computer program controlling the run of AI algorithm. In case of the EA implementation, the overlord applies the operators and it controls whether the goal was reached.

[5]The knapsack problem is one of the classical AI exercises. It belongs into the complexity class NP. Imagine a set of items and a bag with limited weight. Each item has weight and value. The knapsack problem asks what is the most efficient way of loading the bag with given items.

In the case of the knapsack problem, `1` means an item is loaded, `0` means an item is not loaded. Fitness would represent whether the bag is under- or overloaded and how much value is loaded. The GA solution of the knapsack problem is used as an example of PROGA translation to SWI-Prolog (see attachment C).

What remains to be described more precisely are the operators of reproduction and selection.

2.2.1 Operators of reproduction

The definitions of operators of reproduction are inspired by biology. Similar processes take place during the reproduction of biological genome.

There are three different operators of reproduction: two types of crossover and one type of mutation. These reproduction operators are usual and are used in the PROGP and PROGA. It is possible to imagine some other operators.

The first type of crossover is one point crossover. To perform the one point crossover, it is necessary to choose two good (according to their fitness value) parents and select randomly a point in their genome. Then take the beginning of the genome to the selected point of the first parent and connect it to the rest of the genome of the second parent (from the crossover point till the end). The same thing is done with the other genome. Two new offspring are created. This takes inspiration from biology and genotype crossover.

The second type of crossover is a two point crossover. To perform the two point crossover, choose two good parents (according to their fitness value) and select randomly two points in their genome. Then do the same thing as in the one point crossover. When the second crossover point is reached change back to the first genome. Also in this case, two genomes are created.

There is an example of a two point crossover applied to parents with genomes 000000000000 and 111111111111 in Figure 1. The crossover points were selected as 4 and 8.

Clearly the length of the genome is preserved by the crossover operator. This operator also preserves a significant part of the genome with a good fitness to the next generation. There is a good chance that the offspring will have the good properties of its parents.

The operator of mutation applied to a binary vector is called bit mutation. In this case, there is only one parent and one offspring. Bit mutation selects randomly one point in the genome and changes it from 1 to 0 or the other way. Bit mutation also preserves the good properties of the parent (it preserves most of the genome to the next generation) but may lead to the creation

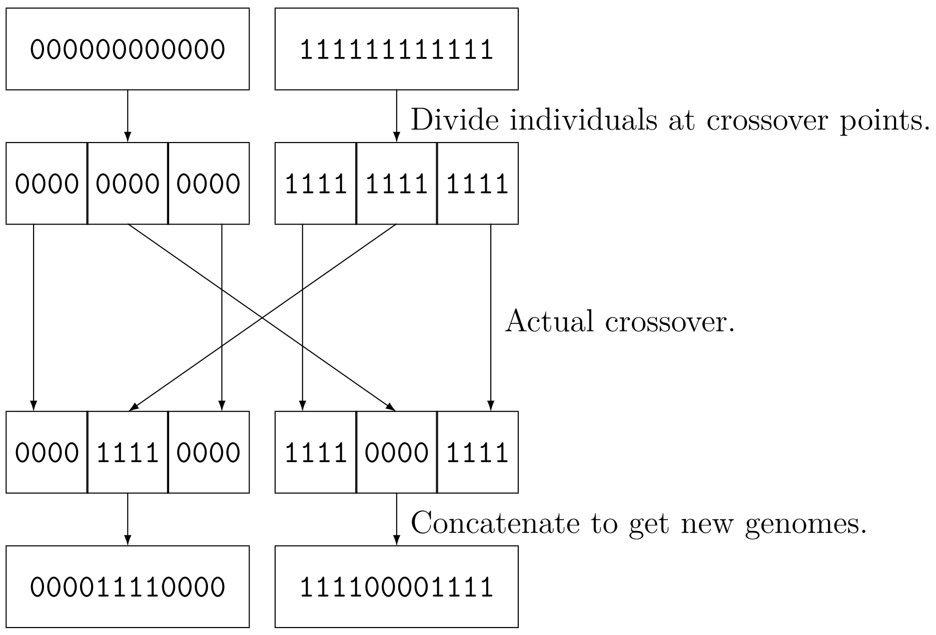

Figure 1: Example of two point crossover

of a completely new genome.

The mutation operator was the probable beginning of biological evolution. The first replicator was a single large molecule. It carried the information about cloning itself. There would be no evolution possible if no mistakes or errors occurred during the replication process. This error is usually called mutation. The mutation of the first replicator could change the replicator's genome and so could create a completely new genome.

These operators are usually used together in EA. Both crossover operators have the tendency to end in local minima if they are not combined with mutation. The mutation operator is a random search algorithm.

2.2.2 Operators of selection

The selection operator decides which genomes are going to survive to the next generation. This operator is the equivalent of the concept of mortality in biology.

Individuals that are not fit enough or are too old die. In the case of EA they are not selected to the next generation.

The simplest idea of the operator of selection would be an algorithm that chooses the genome with best fitness yet. This idea is too simple to work. It would lead to a decrease in population diversity. A low population diversity would increase the probability of ending in some of the local extrema.

One of the possible operators of selection is the tournament operator. It chooses randomly a small part of the population and chooses the best individual in this part. This procedure is repeated until the size of a population is reached.

There are several other possible operators of selection but the tournament operator is the only relevant one for this work.

2.3 Genetic programming

The other possibility for a genome representation (other than a binary vector) is a tree structure. The EA that use tree representation of the genome are called genetic programming (GP). This improvement of GA was first introduced by John Koza [Koza 1992]. It is possible to represent computer programs, mathematical functions, or logic expressions by binary vectors, but it is not very intuitive. A much more natural representation of such structures is a tree.

A mathematical function represented as a tree has operations in nodes of the tree and numbers or variables in its leaves.

The motivation for a tree representation is the predictability of the genome length. Generally, it is complicated to predict the length of a mathematical function or even a computer program. It would be of course possible to make the length of the binary vector variable. In case of more complicated structures (such as mathematical functions), it will be probably necessary to assign more bits of the vector to one term in a genome, which would lead to some difficulties with both operators of reproduction (it will be necessary to control the correctness[6] of the newly created genomes).

The tree representation also allows the definition of simple operators of reproduction for genomes of variable sizes. The operator of crossover for the tree representation chooses randomly one node of each parent and swaps the subtrees generated from these nodes between parents. The crossover defined for a tree representation takes two parents and creates two offspring again.

The operator of crossover used on mathematical functions swaps a subexpression (such that can be put in brackets without effect on the meaning of the function) of one parent for a subexpression of the other parent.

The operator of mutation is more complicated than in the bit vector representation. In the tree representation it is necessary to control whether

[6]There would have to be some sequence that controls whether the genotype has the phenotype representation (whether an application of operator on the level of a genome created an individual).

the chosen mutation point is in the node or in the leaf, and to respect this point type.

The operator of mutation in a mathematical function replaces the number or variable in the leaf with a different number or variable, or it replaces the operation in a node with a different operation. It does not make sense to change the operation in node into a variable.

2.4 Lamarckism

Jean Baptiste Lamarck created the first complete evolutionary theory. It was introduced at the beginning of the 19th century[7]. Lamarck published his work Philisophie zoologique in 1809 and died in 1829. Darwin was born in 1809[8]. Darwin published his work On the Origin of Species in 1859. One part of Lamarck's theory was used to optimize EA. The basic proposition used here is widely known as Lamarckism. But Lamarck's theory is complex and contains many interesting thoughts. Lamarckism is considered disproved by the majority of biologists today. Lamarckism can be briefly described in this way: properties, gathered during a life of an individual, may be passed to its offspring.

According to Lamarckism, a giraffe has a long neck because its parents' necks were stretched during their reaching for high leaves. According to Darwinism, on the other hand, a giraffe with a longer neck has an advantage in the competition with other giraffes and would have more offspring (with longer neck) than giraffes with shorter necks.

How does Lamarckism work in EA? A deterministic search of the space of all possible solutions is added to the normal EA. So we can adjust our description of EA by adding point number 5:

1. Make randomly the first generation of genomes.

2. Evaluate the whole population by a fitness function.

3. Check whether the goal was reached. If so, end with success. If no, continue.

4. Make the next generation:

[7]Jean-Baptiste Lamarck. (2014). In Encyclopaedia Britannica. Retrieved from http://www.britannica.com/EBchecked/topic/328430/Jean-Baptiste-Lamarck [Britannica 2014]

[8]Charles Darwin. (2014). In Encyclopaedia Britannica. Retrieved from http://www.britannica.com/EBchecked/topic/151902/Charles-Darwin [Britannica 2014]

   (a) use an operator of selection on population.
   (b) use an operator of reproduction on population.

5. Go through the population and apply the local search algorithm.

6. Go to point 2.

Applying the local search algorithm has a great potential in shortening the searching time. However, if applied on the whole population, it could have a negative impact on its diversity[9]. To avoid the diversity problem, it is possible to apply the local search algorithm on a part of the population or it is possible to use the so called Baldwin effect. The Baldwin effect uses a local search algorithm but does not replace the genome with its best neighbour, it only changes the genome's fitness function value for the fitness function value of its best neighbour.

2.5 PS2O

The PS2O algorithm is not a usual EA. In fact, it belongs to a different group of BI algorithms. It is an ecology inspired algorithm. Interactions between species and the environment are the subject of ecology [Odum 1977].

In the PS2O algorithm, there are several randomly made species. Each species has a population of a certain predetermined size. The populations are made of genomes. Populations evolve via some evolutionary algorithm. The fitness function rates all genomes of all species, therefore genomes of different species are comparable.

The motivation in biology is clear: this algorithm simulates the evolution process of an ecosystem.

An example of PS2O is an evolution of mathematical functions represented as binary vectors. The species differ in the length of the genome.

It is necessary to define several new operators on the level of species. The first possible operator changes the sizes of species populations. Species with more successful individuals (according to the fitness values) have more individuals than the ones with less successful individuals.

In our example, genomes of certain length would be more successful than genomes of other lengths. The more successful length of genomes would be more probable in the next generation.

Other possible operators change the definitions of species. In some intervals, it is necessary to create a new species. The new species definition

[9]All genomes in the population could be similar. This has of course bad impact on the evolution search.

might be fully random, or it might respect the successfulness of the existing species (such operator must create more often species with similar definitions as successful species).

It is easy to define such an operator for species distinguished by length of binary vectors. It may be impossible to create a function with the demanded property in the first generation, because functions with the demanded property could have different length of the genome that functions included in the first generation. After some generations, it is possible to create new species by choosing a different length of binary vector.

The alteration of the EA looks like this:

1. Make randomly the first generation of genomes of several species.
2. Evaluate the whole population by a fitness function.
3. Check whether the goal was reached. If so, end with success. If not, continue.
4. Use operators defined on the level of species.
5. Make a next generation:
   (a) use an operator of selection on every species population.
   (b) use an operator of reproduction on every species population.
6. Go to point 2.

2.6 Bee colony algorithm

The last group of BI algorithms includes swarm intelligence algorithms. There is typically no overlord in the case of swarm intelligence. No program controls the swarm. The solution space search is the work of a set of simple agents. These algorithms are usually inspired by the life of social insects, such as bees or ants.

The algorithm described here is inspired by a bee colony. One of the problems that a bee colony has to solve is the search for a food source. Bees evolved a special kind of communication for this purpose.

The artificial bee colony algorithm is based on such a communication. The size of the food source represents the quality of the solution of a demanded problem. To learn how good a food source is, it is necessary to count a fitness function.

There are three different types of bees: scouts, employed bees, and onlookers. Scouts search the environment randomly. Employed bees use information gathered by scouts and search deterministically the solutions suggested by scouts. Onlookers share the information and decide where to send the employed bees.

The swarm intelligence algorithm is not in the main focus of this work. It is introduced very briefly, here. More details of such a algorithm might be found in [Binitha et. al. 2012].

3 Logic and EA

3.1 Logical programming

Logical Programming (LP) is a programming language paradigm. LP is based on Horn logic. The language of LP contains predicates, variables, functions, and logical connectives (conjunction and implication)[10].

Every line of a LP program is a Horn clause. Every clause is created by predicates. Every predicate has a name and a set of variables. An example of a predicate follows.

The predicate `pred( V1, V2)` is called `pred` and in its set of variables are two members, which are `V1` and `V2`.

The Horn clause is dividable into a head and a body. The head always consists of one predicate. The body and the head are connected by an implication. Predicates in the body are connected by a conjunction. A Horn clause with only one predicate (it might be viewed as the head) is called a fact.

The clause is supposed to be read this way:

```
If all predicates of the body are true, then the head is true.
```

Usually the implication is directed from the left side to the right. However, it is directed from the right side to the left in the syntax of LP. The head of the predicate is on the left side and its body on the right side of the implication that is directed from right to left. The symbol of implication is '`:-`' and the symbol of the conjunction is '`,`' in the syntax of LP.

An example of a Horn clause in classical logic (every predicate is for simplification written without the set of variables).

$$body1 \;\&\; body2 \;\&\; body3 \rightarrow head1$$

Translation of this Horn clause in the language of LP follows.

```
head1 :- body1, body2, body3.
```

LP program is a database of Horn clauses. A run of LP starts with asking a question. This question is a predicate. The LP tries to prove the question from the database. The proof is searched by Selective Linear Definite (SLD) clause resolution. SLD clause resolution can be seen as the automatic proof searching algorithm. The soundness and completeness are provable for SLD clause resolution in LP. [Nilsson et. al. 2000]

[10]This particular definition of LP language is based on the lectures of RNDr Jan Hric but can also be found in [Nilsson et. al. 2000]

3.2 Prolog

Prolog is a programming language. It is based on the LP paradigm. It was implemented in France in the 1970's. [Bratko 2001]

SWI-Prolog is an implementation of Prolog first designed by John Wielemaker in 1986. It is implemented in C. [Wielemaker 2014]

The language of Prolog is an extension of the language of LP. The language is extended by logic operators cut and negation and one more logical connective, the disjunction. But it is possible to look at disjunction only as the abbreviation for another definition of a predicate. Therefore the disjunction connective is going to be overlooked in the following text.

3.3 Motivation

Creating a theory that suits some finite set of properties is simple. It is enough to choose the finite set of properties as axioms. However, creating a theory that generalises this finite set of properties is not a simple task. It is the same problem as creating a finite theory for an infinite set of properties. The creation of such theories might be indeed very complicated.

An attempt to use GP in a logical programming paradigm might be viewed as an attempt to teach a computer to find such theories. The evolving program is the searched theory in the case of this motivation. The question asked by the program is a property that the theory should respect.

The control of how successful the GP has been is up to the programmer. For example, the control whether the theory solves the demanded task, or whether the theory is complete. The results are of course conditioned by the finite set of properties. The success of evolution searching depends on definitions of the operators.

An attempt to create GP for the logical programming paradigm is presented in [Tang 1998]. This attempt was compared with GP for functional programming and inductive logic programming. However, both attempts at GP for logical programming and functional programming were implemented in programming language Lisp. Inductive logic programming is EA on the Horn clauses. Individuals are represented as strings. GP for logical programming was significantly less successful than the other two approaches. Generally, the GP for logical programming should not have worse results than the other two approaches.

As previously stated, Prolog is an extension of logical programming, so it is more natural to use Prolog as a programming language to implement GP using logical programming. The main task of this work was to design such a program. The resulted program is based on Olsson's Prolog Genetic

Algorithm (PROGA) and is called Prolog Genetic Programming (PROGP). PROGP is designed in SWI-Prolog.

Recall that PROGP is implemented in Prolog. However, PROGP is genetic programming designed for the logical programming paradigm. The logic operators (cut and negation) cannot be used in the evolving programs. Moreover, there is no connective of disjunction.

4 PROGP

4.1 Properties of LP in context of GP

4.1.1 Structure of the program

It is necessary for GP to represent the genome as a tree graph. PROGP uses the left to right, top to bottom orientation of the tree graph.

```
head1 :- body1, body2.
head2.
head3 :- body2.
...
```

Figure 2: LP

Figure 2 shows an example of the LP program.

When creating the graph, the first head of the first clause is the first node. The left successor is the body of the clause. The body of the clause is a list of predicates. If the body is empty (the head was a fact), the list of predicates is empty and an empty node is added to the graph. The right successor is the head of the next clause, etc.

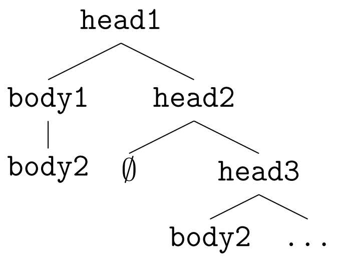

Figure 3: Tree representation of LP

An example of the tree representation of the LP program is shown in Figure 3. Such graphs have some nice properties. The graph is always connected and it is not cyclic (it is a tree). The graph grows wider only on the right side. There is a path through the graph that is made only of binary nodes, except the last one which is unary. Nodes that are not on this path are all unary.

The Prolog representation of such a graph is simple. The representation is a list of tuples. [Bratko 2001]

Figure 4 shows an example of the Prolog representation of the tree from Figure 3.

```
[ [head1, [ body1, body2] ], [head2, 'fact'],
          [head3, [ body2 ] ], ... ]
```

Figure 4: Prolog representation of tree representation of LP

The node is the first member of the tuple and it is a predicate. The left successor is the second member of the tuple and it is a list of clauses. The atom 'fact' is used in the case that the list should be empty. The first member of the next tuple is the right successor.

PROGP is GP since the representation of the LP program is a tree.

4.1.2 Variables

Until now there was no need to talk about the sets of variables of predicates. Predicates (both their names and sets of variables) have been substituted by a predicate name in every example of LP code.

Every predicate has its arity in Prolog and LP. The arity sets the number of variables that the predicate uses. For simplicity reasons[11], only predicates with the same arity in PROGP are used. That is possible because it is simple to add anonymous variables (such that do not affect the value of the predicate) to the sets of variables that the predicate uses. A translation algorithm (see attachment B) expands the Prolog program to one that has only predicates with the same arity.

Every set of variables in PROGP should be clearly represented as Prolog varable type list. However it is represented as the Prolog variable type atom. Prolog allows simple manipulation between the Prolog variable type atom and list[12]. The atom type is used because it allows easier manipulation with the genome. The translated list form allows better mutation handling and easier translation between genome and individual.

The arity of predicates has to be decided before PROGP run. It is simple to decide the arity for such simple tasks as are presented in the section 6. It might be necessary to use higher arity than seems to be necessary for some more complicated tasks.

4.2 PROGP vs. PROGA

PROGA, as introduced by B. Olsson [Olsson 1996], is an implementation of the classical genetic algorithm designed in SICSTUS Prolog. The genome

[11] mostly because of the correctness of operators (see section 2.3).

[12] The translation is possible because of the Prolog built-in predicate `term_to_atom/2`. This predicate unifies the atom with a term and in reverse.

is in this case a binary vector. It uses the two point crossover as an operator of reproduction and the tournament or the roulette as operators of selection. The program consists of 6 files and 187 clauses. There is an example of PROGA (translated into SWI-Prolog) applied on the knapsack problem (see attachment C).

PROGP is the alteration of PROGA. It is an implementation of genetic programming for the logical programming paradigm. It was designed in SWI-Prolog. There are 177 clauses in two files. The program's description and examples of usage follows.

4.2.1 Alteration of PROGA

First it was necessary to translate PROGA to the SWI-Prolog language. This translation is presented in the example of the EA solution of the knapsack problem. Although there is a library in SWI-Prolog that should make all SICSTUS code readable, there have still been some problems with PROGA translation. The actual translation does not use this library and the PROGA differs from the result in only a few predicates definitions.

The main difference between the PROGA and PROGP is in the representation of the genome. No part of PROGA that operates with genomes could be used in PROGP. That includes the operators of reproduction (`xover/9` and `mutation/10`). Of course, the original fitness function could not be used nor could the Lamarckian evolution (`fitness/3` and `lamarckian_evolution/7`). The operator of initialization also had to be altered (`CreateGraphs/6`).

Some parts of the program were altered and reduced to make the code more readable (`initialise/15` and `showPop/7`). The user interface may be less friendly because of such alterations. On the other hand, PROGP is much shorter and it has even fewer clauses than PROGA (even though PROGP operates with more complicated structures). The operator of selection (`select/8`; although it has been reduced to only the tournament operator) and the main cycle (`ga/0` and `run/14`) are originals from PROGA or with as little changes as possible.

4.3 PROGP description

PROGP consists of two files. The main one is GP.pl. This file contains the main cycle and most of the program. The second file is called fitness.pl. It is automatically consulted by GP.pl. It is necessary to alter the fitness.pl file for every alteration of PROGP on a specific problem.

Several text files are created during the run of the program. All these files are stored in the same folder as the program. These files contain details on the evolution process. The results of the evolution themselves are discussed later (see section 6).

The statistics of SWI-Prolog run are written on the screen after the creation of every new generation. These statistics contain the actual parameter definitions, detail of SWI-Prolog memory usage and the time consumption of the whole run.

Predicate `ga/0` starts the evolution process.

5 Source code of PROGP

The source code is widely commented. This section allows easier understanding of PROGP's principle without the technicalities. For example, the predicates operating with the Prolog variable types are omitted. However, all evolution-related predicates are mentioned and introduced here.

The only exception are two evolution-related predicates that are very technical and the description of their run is omitted. The first is called `createGraphs/6`, the second is called `mutation/10`. These predicates operate with the structure of genome. This is the cause of the technicalities. This section describes their impact on the genome.

There are two predicates that are placed in the file fitness.pl the first is `getPars/11`, the second is `fitness/3` (almost everything that is called by this predicate is placed also in the fitness.pl file).

The evolution starts via calling the predicate `ga/0`.

The evolution of a certain existing population starts via calling the predicate `runExistingPop/1`.

5.1 General idea of PROGP

Recall the general algorithm of EA with Lamarckism applied. PROGP of course respects this general scheme.

1. `initialise/12` makes randomly the first generation. `run/14` starts the main cycle of the program.
2. `evaluate/3` runs the fitness function on every member of the population.
3. `showPop/7` checks whether the goal has been reached. If so, it ends the program with success. If not, it lets the program continue.
4. To make the next generation:
   (a) `select/8` runs operator of selection.
   (b) `xover/6` and `mutation/10` run operators of reproduction on population.
5. `lamarckian_evolution/7` goes through the population and applies the local search algorithm.
6. `run/14` starts recursively in point 2 again.

This is the general idea of PROGP. Every predicate mentioned here is described in detail later.

The PROGP has one little alteration in the code that differ such general idea. The mutation operator is used after the operator of selection. The number of mutated genomes in every population respects the parameter of mutation probability (see section 5.3.1).

5.2 ga/0

```
ga/0 :-
    initialise/12,
    evaluate/3,
    run/14.
```

The first generation is created by `initialise/12`. The population is a list of tree representations as introduced in section 4.1.1.

The fitness function is applied on every genome via `evaluate/3`. This is done by predicate `fitness/2`. The description of fitness function source code is described in section 5.5. Some examples of fitness functions are described in section 6.

The main cycle of PROGP is called `run/14`.

5.2.1 initialise/12

The predicate `initialise/12` is called by `ga/0`.

```
initialise/12 :-
    getPars/11,
    createGraphs/14.
```

The parameters necessary for the run of PROGP are loaded from fitness.pl by predicate `getPars/11`. The predicate is in detail described later (see section 5.3.1).

Predicate `createGraphs/6` works straightforwardly, but it is long and fairly complicated. Details of individual steps are commented in the program. This predicate creates a population. The created population is a list of programs. Every program respects the representation as described earlier 4.1.1.

5.3 runExistingPop/1

```
runExistingPop/1 :-
    loadPopulation/4,
    getPars/11,
    run/14.
```

Running PROGP from the existing population is possible via `runExistingPop/1`. The population needs to be stored in a specific way. It is possible to start with the population created by PROGP (the text file 'gen N.txt' where N is a number of a given generation). To start with the user defined population, it is necessary to create a file that respects the official created population (including the objective and fitness values).

The parameters necessary to run PROGP are loaded from the file fitness.pl by the predicate `getPars/11`. The predicate is described in detail later (see section 5.3.1).

The main cycle of PROGP is called `run/14`. Its detailed description follows (see section 5.4).

5.3.1 getPars/11

The predicate `getPars/11` is called by `initialise/12` and `runExistingPop/1`. The predicate `getPars/11` is placed in the file fitness.pl.

The predicate `getPars/11` is responsible for the loading of 11 parameters necessary for PROGP's run.

The first parameter is called `ShowTime` and it is an integer. It refers to the predicate `showPop/7`. Parameter `ShowTime` sets how often a generation is saved as a callable file (of the format 'gen N.txt' where N is a number of a given generation).

The second parameter is called `Generations` and it is an integer. It sets the maximum amount of generations. Parameters `Generations` and `ShowTime` are related. If the parameter `ShowTime` is not lower than `Generations`, no callable generation is saved. It is recommended to choose the `ShowTime` as divisor of the `Generations`.

The next two parameters refer to the list of variables used with each predicate (as discussed in section 4.1.2.). The first of these parameters is called `NrVar` and it is an integer. It sets how many different variables can be used in the genome. The number is usually chosen as lower than 10.

The next parameter is called `ListVar` and it is a list. Each member of the list is `'Int'` or `'List'` (with the simple quotes, it is a Prolog type atom). The `ListVar` parameter sets both the length of the variables list and each member type (integer or list).

The fifth parameter is called `NrPredicates` and it is an integer. It sets how many predicates can be used in the program. This number also sets how long one line of the program can be (every predicate can be used only once in the predicate's body). Maximal length of a line is `NrPredicates` + 1.

The sixth parameter is called `NrChromosomes` and it is an integer. It sets the size of the population.

The seventh parameter is called `MaxDepth` and it is an integer. `MaxDepth` sets the maximal number of lines in the program.

The next two parameters refer to the reproduction operators. The first is called `TSize` and it is an integer. `TSize` sets the size of the tournament (how many genomes are going to compete with each other). `TSize` must be lower than `NrChromosomes`. The second is called `Xnr` and it is an integer. `Xnr` decides how many runs of the crossover operator there will be. Each run of the crossover creates two new genomes. That means that the `Xnr` must be lower than half of the `NrChromosomes`[13].

The tenth parameter is called `MutP` and it is a real number from the interval [0,1]. `MutP` it sets the probability of performing a mutation on a genome.

The last parameter is called `LamP` and it is a real number from the interval [0,1]. It refers to the probability of performing Lamarckian evolution on a genome.

There are two groups of these parameters. The first group contains parameters that cannot be changed while solving a certain task, or parameters that do not affect the actual evolution process. This group contains `ShowTime`, `Generations NrVar`, `NrPredicates`, `ListVar`, and `MaxDepth`. These predicates affect the size of the searched space.

The second group of variables contains the parameters that affect the evolution directly. These include the parameters `TSize Xnr`, `MutP`, and `LamP`. The parameters define how much of the search space would be searched in one generation.

The test of the effect of the second group of parameters on the evolution process is presented in section 6.3.1.

5.4 run/14

The predicate `run/14` is called by `ga/0` and `runExistingPop/1`.

[13]Recall that every application of the crossover operator creates two offspring (see section 2.2.1).

```
run/14 :-
    showPop/7,
    reproduce/12,
    mutation/12,
    lamarckian_evolution/12,
    run/14.
```

Predicate `run/14` represents the main cycle of PROGP. This predicate is recursive.

The operators of selection and crossover are called by predicate `reproduce/12`.

The mutation operator is called by predicate `mutation/12`.

The Lamarckism is applied by predicate `lamarckian_evolution/12`.

5.4.1 reproduce/12

The predicate `reproduce/12` is called by `run/14`.

```
reproduce/12 :-
    xovers/10,
    evaluate/3,
    fillup/12.
```

5.4.2 xovers/10

The predicate `xovers/10` is called by `reproduce/12`.

```
xovers/10 :-
    select/8,
    select/8,
    xover/6,
    xovers/10.
```

The operator of selection (the tournament operator) is called via the predicate `select/8`. The operator works straightforwardly as described in 2.2.2. Details about its code are described in the source code comments.

The predicate `xover/6` creates two new genomes using the crossover operator. Two definitions of crossover are used in PROGP. The first is called `xovera/9` and it exchanges the subtrees of a chosen node (see Figure 5). The second is called `xoverb/6` and it exchanges the bodies from a chosen node (only left successors in the tree representation) (see Figure 6)[14].

[14]The predicate `xoverb/6` might be considered as special application of two point crossover. The first point is the selected node. The second point is the right successor of selected point.

The second definition of crossover was added while testing PROGP for optimizing the evolution results.

Predicate `evaluate/3` has been described before.

Predicate `fillup/12` fills up the new population to the predetermined population size with the fittest members of the previous population. The operator of selection is used for choosing the fittest members.

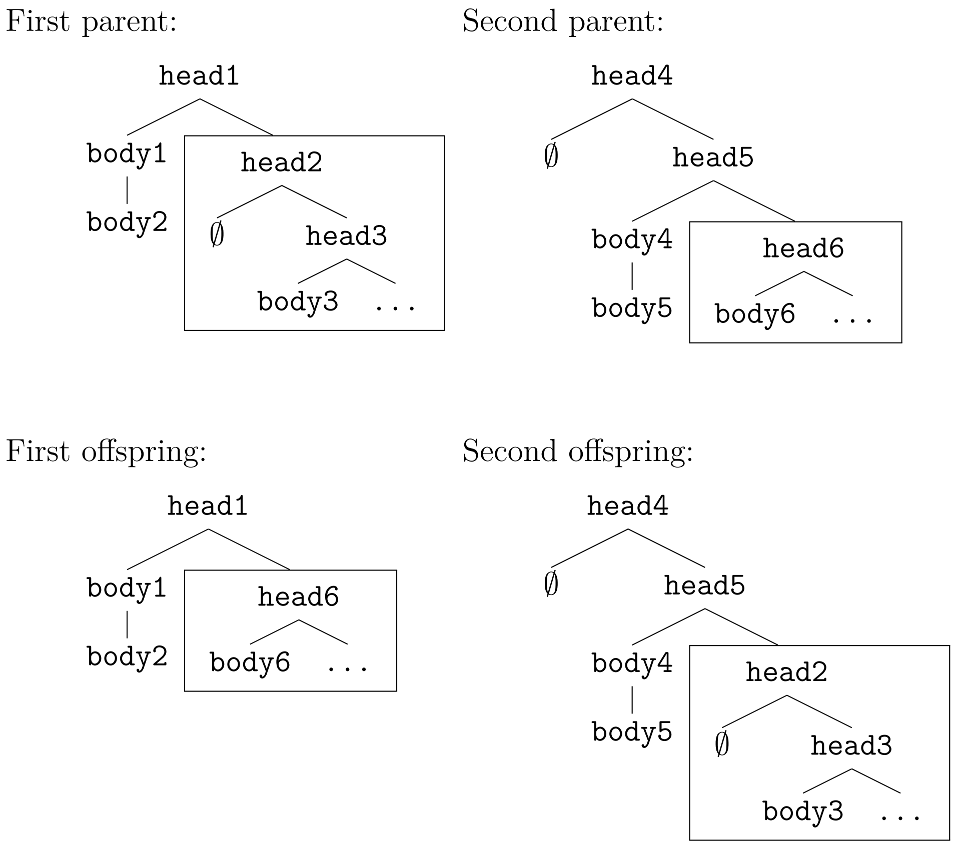

Figure 5: Example of `xovera` aplication

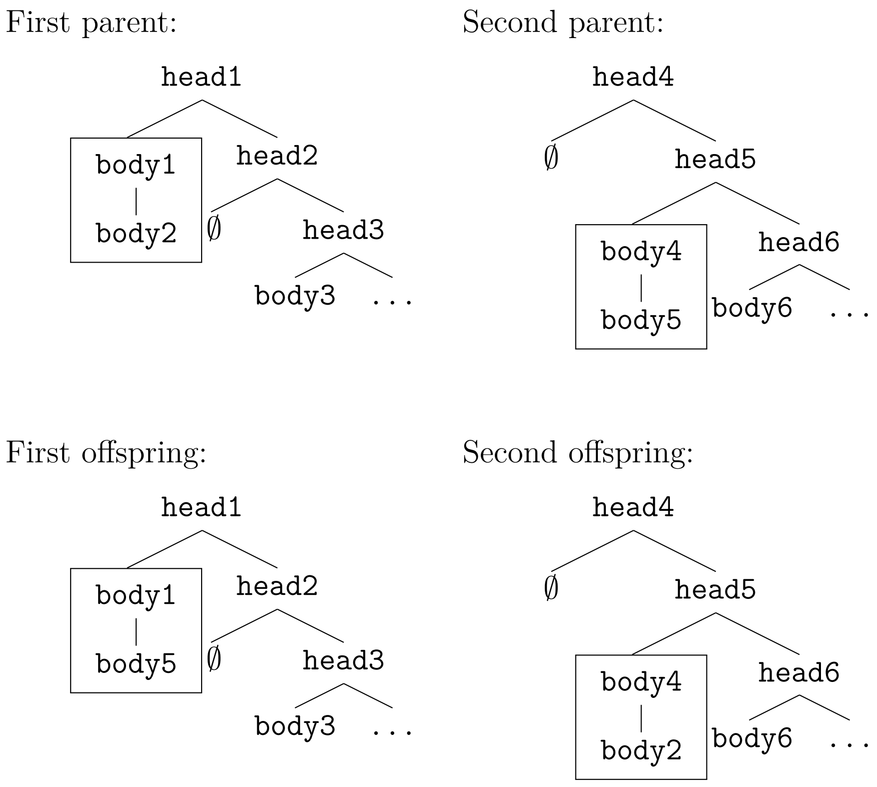

Figure 6: Example of `xoverb` aplication

5.4.3 mutation/10

The predicate `mutation/10` is called by `run/14`.

```
mutation/10 :-
    mutate/5,
    fitness/3,
    mutation/12.
```

The predicate `mutation/10` mutates with a predetermined probability a member of the population. The predicate is recursive. It works straightforwardly but is complicated. The process is commented in the program in detail.

It is possible to mutate one of the existing predicates in the program. To mutate a predicate means to create a new predicate and exchange it for a chosen one.

Other possibility is to mutate one of the variables from a set of variables. First a set of variables is chosen, then the actual variable to mutate is selected. In the case of integer type variable, a new variable is created. In the case of a list type variable, it is possible to create a completely new list variable, but it is also possible to divide the list and change one half of it and preserve the other[15].

5.4.4 lamarckian_evolution/7

The predicate `lamarckian\_evolution/7` is called by `run/14`.

```
lamarckian_evolution/7 :-
    findall( permutation/2, or subSet/2,
    evaluate1/4,
    lamarckian\_evolution/7.
```

The deterministic space search is provided by predicate `Lamarckian_evolution/7`. The program decides whether to try all possible permutations[16] of the program lines or whether to try some of possible subsets[17] of the program lines. The decision depends on the length of the genome: if the length is higher than 5, then it tries the subsets. The reason for these numerical limitations is the factorial complexity of finding all permutations of bigger lists[18].

The fittest permutation or subset is sent to the next generation (via the predicate `evaluate1/4`).

The Lamarckian evolution as defined here is easily extendible on the Baldwin effect. The individual would not be replaced with the fittest possible neighbour, only the fitness value would be improved.

[15]Variable type list has in Prolog the structure of a tree (every node has zero, one, or two successors. Zero successors in case of list containing an atom. One successors in case of a list containing one another list. Two successors in case of Prolog head/tail representation of list). In case of mutation as defined in PROGP it is possible to create a completely new tree, or it is possible to look at the root of the tree and change one of its successors.

[16]The predicate `permutation/2` is a built-in SWI-Prolog predicate that unifies permutation of a list.

[17]The predicate `subSet/2` is a defined in PROGP and it unifies some subsets of a list. It is not all subsets, because it unifies only subsets of items that follows each other (i.e. for a three item list: `[1,2,3]`, predicate `subSet` does not find a subset `[1,3]`, because item `3` does not directly follow item `1`).

[18]For a set of length 8, there is 37 unifications of predicate `subSet/2` and 40320 unifications of predicate `permutation/2`.

5.5 fitness/3

The predicate `fitness/3` is called by `evaluate/3`. The predicate `fitness/3` is placed in the fitness.pl file.

```
fitness/3 :-
    shapeError/2,
    curGraph2pl/2,
    loadInputs/4,
    runError/6,
    TaxManager/5.
```

The predicate `fitness/3` translates a genome to the individual (translates the genotype to the phenotype) and evaluates individuals based on their objective and fitness value. These values are related. The fitness value is counted from the objective value. The objective value is a sum of all errors that are recognised by the predicate `fitness/3`. Usually the fitness value is in the interval [0, 1] where 1 means that the genome is the desired result in EA.

5.5.1 shapeError/2

The predicate `shapeError/2` is called by `fitness/3`.

Predicate `shapeError/2` tests the properties of the genome (for this test it is possible to use a non-translated genome). First, it is the length of the genome. Length is tested by the predicate `lengthError/2`. A shorter genome is better than a longer one. The predicate `TypeCheck/2` examine the number of facts in the genome and number of recursively called predicates in the genome.

5.5.2 curGraph2pl/2

The predicate `curGraph2pl/2` is called by `fitness/3`. It is placed in the file GP.pl.

For other tests of the individual, it is necessary to translate the genome of the individual to a form which would allow Prolog to load it into the current database. Predicate `curGraph2pl/2` is such a translation. This predicate creates a list of lines of an LP program from the tree representation.

5.5.3 loadInputs/4

The predicate `loadInputs/4` is called by `fitness/3`.

Predicate `loadInputs/4` unifies the I/O sets. The first two are designed for testing the programs only with `True` or `False` outputs. These I/O sets are lists of atoms. Every atom has the form of an input of arguments for the predicate. There are no variables in the argument list. The first of these two sets is the positive I/O set. It contains input that should end with `True`. The second is the negative I/O set. It contains input that should end with `False`.

The other two are designed for testing programs that have other than True or False output for the given input. This I/O set is a list of lists. Every member of the I/O set has two members. The first member is the input argument for the predicate. There is one variable in this input. The second member is the output that should unify with the variable in the input after the program has run. Again, the first of these sets is the positive I/O set and the other is the negative I/O set.

5.5.4 runError/6

The predicate `runError/6` is called by `fitness/3`.

Predicate `runError/6` loads the individual (the translation of the genome) into the database. Then it tries to run the evolved program for every input from the I/O sets. It sums the errors gathered during this run.

A very important part of this predicate is predicate `unloadChrom0`. This predicate unloads individuals from the database of PROGP. This is problematic and results in fails in the run of PROGP[19]. This failure is further discussed later (see section 7.5).

5.5.5 taxManager/5

The predicate `taxManager/5` is called by `fitness/3`.

Predicate `taxManager/5` sums all the gathered errors in `ObjVal` (the objective value) and counts `Fitness` (the fitness value) from it.

The fitness value is counted from the formula $1/(ObjVal + 1)$. It is not possible to compare the masses of fitness values because of this formula. That is the reason for using objective value in the charts in the section 6. The formula of course preserves the order of individuals (for every two individuals if one has a higher objective value than the other, it has a lower fitness value).

[19]the only red cut of PROGP is used while calling of this predicate.

6 Run of PROGP

The run of PROGP is demonstrated on three different tasks in this section. Problems of increasing complexity were chosen. The first task is called the First Member. The resulting theory says whether an item is a first member of a list. The shortest solution has one predicate and it is a fact. The second task is an extension of the first one. It is called the Member. The resulting theory says whether an item is a member of a list. The shortest solution has one fact and one recursive predicate. The third task is taken from [Tang 1998]. It is called Concatenate. The resulting theory says whether a third list is a concatenation of two lists. The shortest solution has one fact and one recursive predicate. It is more complex because there is one more variable added.

Details on how to experimentally measure the success of PROGP were taken from [Tang 1998]. The results are measured in 20 runs of PROGP. The parameter settings were tested manually. PROGP was successful in 50 generations only for the First Member task (see section 6.1). Only one run of the Member task was successful. The rest of the runs were not successful even in 100 generations (see section 6.2). This section also contains a further test of time consumption of PROGP and examples of successful applications of operators of reproduction (see section 6.2.1). The task Concatenate was never successful in 100 generations (see section 6.3). This section also contains a description of the experimental test of parameters (see section 6.3.1).

All I/O sets were defined randomly to fit the demanded solution.

The source code of all three tasks can be found in the attachement (see attachment C).

Five runs out of 20 failed in case of PROGP for the task First Member. Two runs out of 20 failed in the case of PROGP for the task Member. Two runs out of 20 failed in the case of PROGP for the task Concatenate. to sum it up 9 runs out of the total number of 60 failed. A failed run means that PROGP ended with a Prolog error. The error that occurred was a not very well documented error called Signal 22. It should have some connection to the predicate retract/1[20].

6.1 First member

The size of the positive I/O set was 12. The size of the negative I/O set was also 12. There were also 2 positive and 2 negative Input sets that were used

[20]SWI-Prolog built-in predicate `retract/1` removes the clause from the current Prolog database.

to test the individual's output. The highest possible number of countable errors is 28[21].

PROGP was set to run for 50 generations. It could use up to 5 variables and 2 predicates. Every set of variables has 2 items and every item might be a list. The maximum length of a genome in the first generation was 5 (an individual in the first generation can have only 5 lines of LP code). There were 200 individuals in every generation. The size of the tournament was 10 and the operator of crossover was applied 40 times. The operator of mutation was applied with the probability of 0.5 and the operator of Lamarckism with the probability of 0.1.

Every genome with length higher than 5 was disadvantaged by the fitness function. If the application of the fitness function on the genome took more than 0.1 seconds, the genome was also disadvantaged by the fitness function[22].

In the first generation the average objective value was 14.32. The best individual had the objective value 0.05. That means, there was an individual with the demanded property in one of the first generations. The worst individual objective value was 18. Half of all runs were successful in the 8th generation. The last solution was found in the 50th generation. The second last solution was found in the 28th generation.

The Figure 7 describes an average development of the objective value in successful runs of the First Member task. For every generation there is an average objective value of the best individual in every run, the average objective value of all individuals, and the average objective value of the worst individual in every run.

The creation and evaluation of the first generation takes approximately one second. PROGP time demands are larger for every following generation (see Figure 7 or it is visible in every other graph with time measurement). There might be two reasons for such behaviour. The first reason might be the recursive main cycle of PROGP and Prolog's recursion handling. The second reason might be higher time demands on the fitness function for every next generation. More remarks about this behaviour are included in the next section.

The best individuals were always created by the operator of mutation or the operator of Lamarckism. Details about the reproduction operators are described in sections 2.2.1 and 2.4.

[21]Without counting the errors gathered by predicate `shapeError/2`.

[22]These properties are called length and time control in the following section. Application of such properties was proposed in [Tang et. al. 1998].

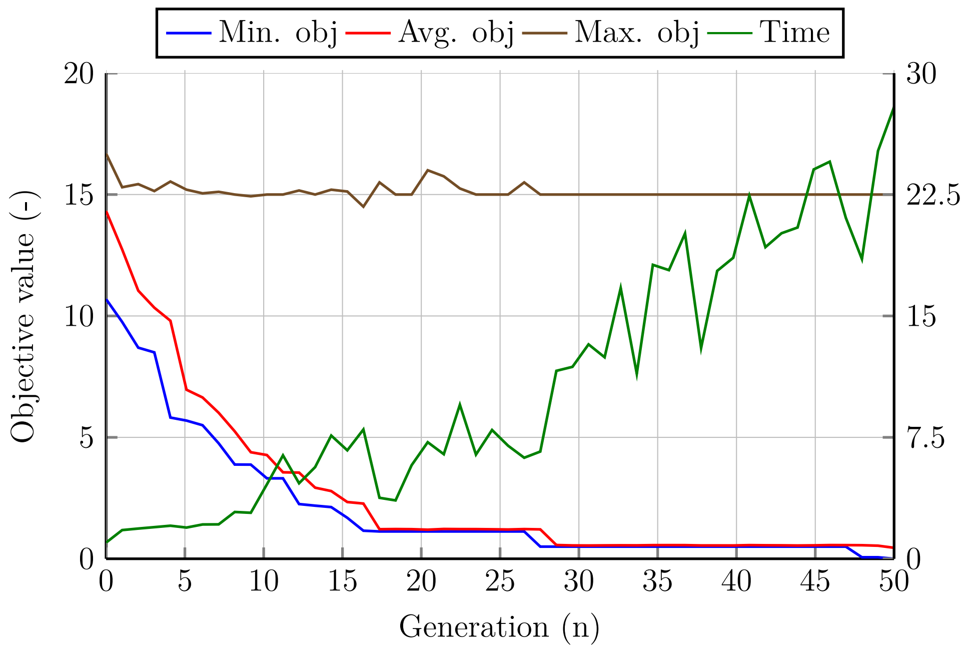

Figure 7: First member

6.2 Member

The size of the positive I/O set was 11. The size of the negative I/O set was 10. There were 3 positive and 2 negative Input sets that tested the individual's output. The highest possible number of countable errors is 26[23].

PROGP was set to run 100 generations. Up to 3 variables and 1 predicate could have been used in the program. Every variables list had 2 items. The first item was an integer. The second item was a list. The maximum length of individual in the first generation was 6. There were 200 individuals in every generation. The size of the tournament was 6 and the operator of crossover was applied 30 times. Operator of mutation was applied with probability 0.8 and operator of Lamarckism with probability 0.05.

The length and the time control was the same as in the First Member task.

The Figure 8 describes an average development of the objective value in successful runs of the Member task. The chart respects the setting from the First Member task.

It is interesting to note that every significant improvement of the minimal objective value took place before the 50th generation.

[23]Without counting the errors gathered by predicate `shapeError/2`.

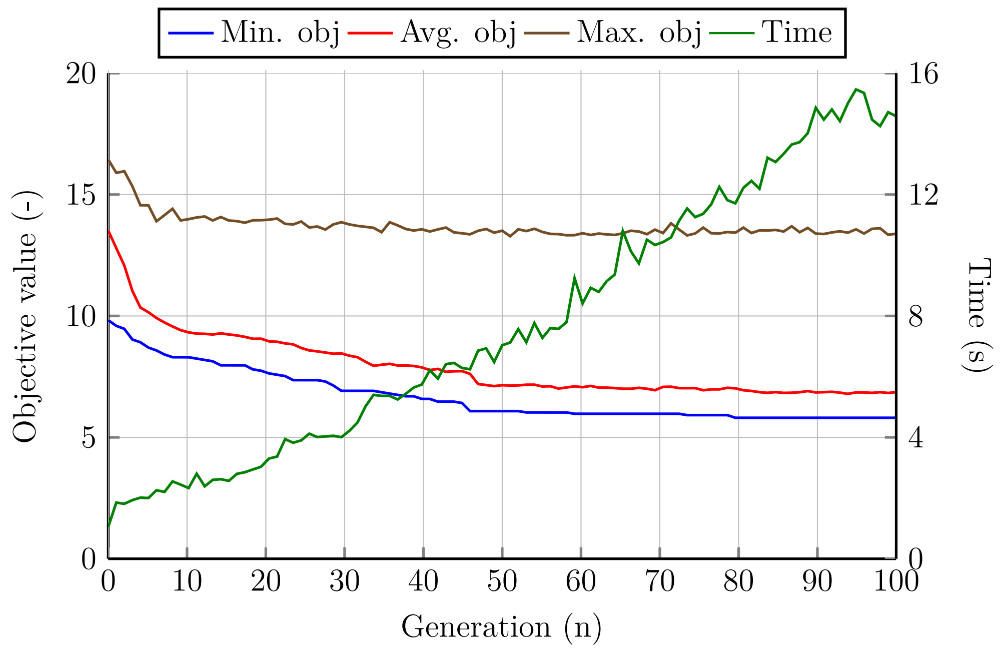

Figure 8: Member

The time demands are lower than in the First Member task. The difference is very high. The First Member time demands are almost twice as high in the 50th generation as those of Member in the 100th generation. The time consumption depends on settings and the size of the I/O sets. The most significant is the difference in the Lamarckian evolution probability and the number of crossover applications.

Every run that successfully reached the 50th generation was started again with the same settings from the 50th generation with the same settings. It is possible to compare the time spent on both possible runs in Figure 9. It is clear that the higher time demands are caused mostly by Prolog's recursion handling (because the curve of time spent during the original 20 runs is very similar to the curve of time spent on the runs from the 50th generation).

This particular result might be used to improve PROGP. It might be possible to lower the time demands by using a non-recursive main cycle (see section 7.4).

6.2.1 Genome evolution example

Only one run from all 20 has found the winner. The evolution of the winner is described in this section. The genomes are translated into individuals.

Two of the output files created by running PROGP are used in this sec-

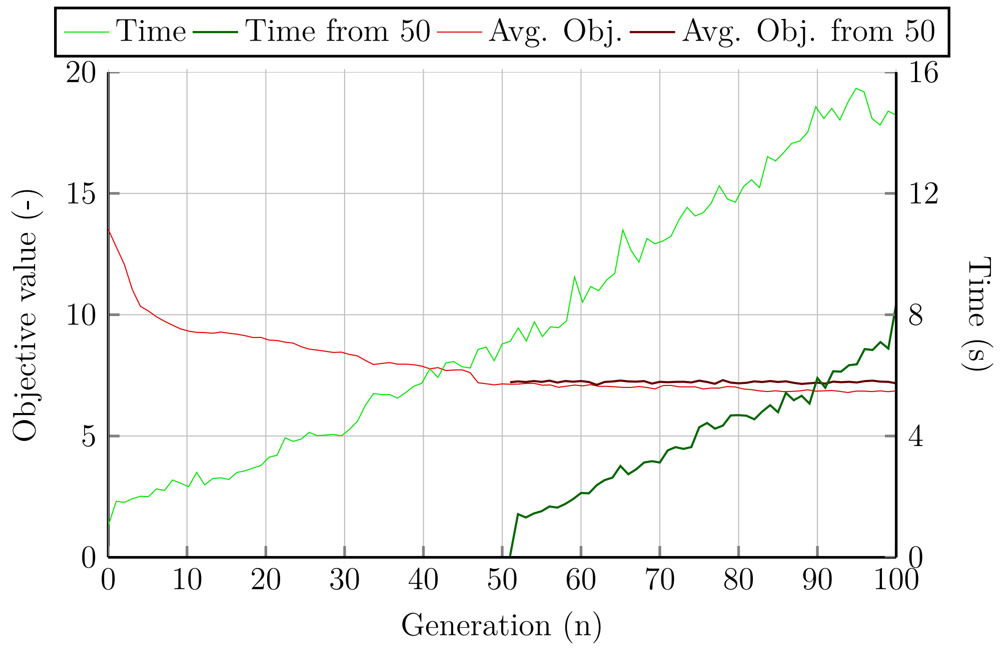

Figure 9: TimeMember

tion. The first is called `results_parental_tree.txt`. This file contains details of how every genome was created. The second output file is called `results_pop.txt`. This file contains all populations of all generations, including the time spent on every population and the objective and fitness values for every genome.

The first described individual was created in the 1st generation and its objective value was 11.02.

```
pred0( V3, V2) :- pred0( 3, [ 9]).
pred0( V0, [ V0]).
```

In this particular run, it was not the best individual in the first generation, but the best individual from the first generation did not survive to the second generation. In the second generation, 11.02 is the minimal objective value. The average objective value was 13.5 in the first generation. The maximal objective value was 17.06. The following genome was created in the 5th generation.

```
pred0( V3, V2) :- pred0( 3, [ 9]).
pred0( V0, [ V0| V1]).
```

Its objective value was 10.02 and it was the minimal objective value of the generation. The average objective value of this generation was 11.59.

The genome was created by the mutation operator from the first described genome. The mutation took place in the variables list of the head of the second line. The third described genome was created in the 8th generation.

```
pred0( V2, V2) :- pred0( 3, [ 9]).
pred0( V0, [ V0| V1]).
```

Its objective value was 9.02. It was again the minimal objective value of the generation (another genome with objective value 9.02 was also found in the 7th generation). The genome was again created by mutation from the previously described genome. The mutation took place in the list of variables of the head of the first line. The fourth described genome was created in the 16th generation.

```
pred0( V2, [ 10| V1]) :- pred0( V1, [ 9]).
pred0( V0, [ V0| V1]).
```

Its objective value was 9.02. It was the minimal objective value of the generation. There were three mutations applied on the third described genome to create the fourth, although it differs only on two positions of the first line.

The next operator used in this evolution was the operator of crossover. It was the operator with definition `xoverb`. This crossover was applied in the 21st generation. The first two lines of the table contain the parents, the third and the fourth line of the table contain the offspring.

1st parent	`pred0( V2, [ 10| V1]) :- pred0( V1, [ 9]).`
	`pred0( V0, [ V0| V1]).`

2nd pratent	`pred0( V3, [ V1| 10]) :- pred0( 2, [ V0| 8]).`
	`pred0( V0, [ V0| V2]).`

1st offspring	`pred0( V2, [ 10| V1]) :- pred0( 2, [ V0| 8]).`
	`pred0( V0, [ V0| V1]).`

2nd offspring	`pred0( V3, [ V1| 10]) :- pred0( V1, [ 9]).`
	`pred0( V0, [ V0| V2]).`

The evolutionary process of the first parent is described earlier. The second parent was created in the 18th generation and its objective value was 9.02. The evolution of the first offspring follows. Its objective value was 9.02.

The next operator used in this evolution was a crossover again. However, it was the operator with definition `xovera`. This crossover was applied to the 28th generation. The table composition is the same as in the previous crossover.

1st parent	`pred0( V2, [ 10| V1]) :- pred0( 2, [ V0| 8]).`
	`pred0( V0, [ V0| V1]).`

2nd pratent	`pred0( V3, [ 4| []]) :- pred0( 2, [ 2|V0]).`
	`pred0( V0, [ V0| V2]).`

1st offspring	`pred0( V3, [ 4| []]) :- pred0( 2, [ 2| V0]).`
	`pred0( V0, [ V0| V2]).`
	`pred0( V0, [ V0| V1]).`

2nd offspring	`pred0( V2, [ 10| V1]) :- pred0( 2, [ V0| 8]).`

The evolutionary process of the first parent was described earlier. The second parent was created in the 23rd generation and its objective value was 9.02. The evolution of the first offspring follows. Its objective value was 9.03 and it was created in the 28th generation. In the same generation a mutation was applied to the first offspring. This genome is the seventh genome described here.

```
pred0( V3, [ 4| [] ]) :- pred0( 2, [2| V0]).
pred0( V0, [ V0| V2]).
pred0( V0, [ 0| V1]).
```

This genome had the objective value 7.03. As mentioned earlier, it was created in the 28th generation. The mutation took place in the variables set in the head of the third line. The value 7.03 is the minimal objective value of the 28th generation. The eighth described genome was created in the 32nd generation.

```
pred0( V3, [ V3| [] ]) :- pred0( 2, [2| V0]).
pred0( V0, [ V0| V2]).
pred0( V0, [ 0| V1]) :- pred0( V2, [[]] ).
```

Its objective value was 7.03. It was once again the minimal objective value of the generation. The genome was created by two mutations from the previously described genome. The mutations took place in the list of variables of the head of the first line and on the second predicate of the third line. The ninth described genome was created in the 35th generation.

```
pred0( V0, [ V0| V2]).
pred0( V0, [ 0| V1]) :- pred0( V2, [[]] ).
```

Its objective value is 7,02. It was the minimal objective value of the generation. The chromosome was created by Lamarckism, which cut the first line out. The tenth described chromosome was created in the 41st generation.

```
pred0( V0, [ V0| V2]).
pred0( V0, [ V2| V1]) :- pred0( V2, V1 ).
```

Its objective value was 6.02. It was the minimal objective value of the generation. The tenth genome was created by two mutations of the ninth. The eleventh genome was the winner.

```
pred0( V0, [ V0| V2]).
pred0( V0, [ V2| V1]) :- pred0( V0, V1 ).
```

It was again created by mutation.

6.3 Concatenate

6.3.1 Experimental settings searching

It is necessary to set the parameters of evolution before every run of PROGP. Is there a single setting of PROGP that solves any problem in the most efficient way? If there is no such setting, then it is possible to extend PROGP to a PS2O algorithm and let the evolution of species decide the best setting (see section 7.3). However, the answer to this question is a task for future work.

3 different settings were compared by an experiment in this section. The comparison takes into account 2 values, the minimal objective value and the run-time of every run. The 3 compared settings were chosen only as examples and there might be a more efficient setting.

There are parameters that cannot be changed. These are the parameters that decide how often the population is saved, the number of generations, the number of variables used, the list of variables, the number of predicates used, the size of the population, and the maximal length of a genome in the first generation. All these parameters are set the same way for all three settings.

Every 50^{th} generation is saved, there are 50 generations, up to 5 variables can be used, the list of variables contains 3 lists, there might be 2 predicates used, the size of a population is 200, and the maximal length of the genome in the first generation is 6. These parameters cannot be changed because they are problem specific (the list of variables) or because the comparison would not make sense. The three different settings description foolows

The I/O sets were defined the same way for all three settings. The positive I/O set had 8 examples, the negative I/O set had 5 examples. Because the program Concatenate would be probably more often used for searching other output than True or False. Both the positive and negative input sets that wait for the output have 4 examples. The highest possible number of countable errors is 21[24].

The first of the values that might be different for every setting was the size of the tournament. But it was set to be 10 in every of the three settings.

The second value that actually differs amongst the settings was the number of crossovers made in every generation. In the first setting it was 20. In the second it was 40. In the third setting it was 60.

The third value also differed amongst the settings. It is the probability of mutation. In the first setting it was 0.7. In the second setting it was 0.5. In the third setting it was 0.8.

The last parameter that was different for each setting was the probability of Lamarckian evolution. In the first setting it was 0.01. In the second setting it was 0.1. In the third setting it was 0.3.

The parameters in the second settings were the same as in the First Member task.

The experimental measurement was applied on five runs of the three settings described earlier. PROGP was run for the same random population (the first population was the same in all 5 runs).

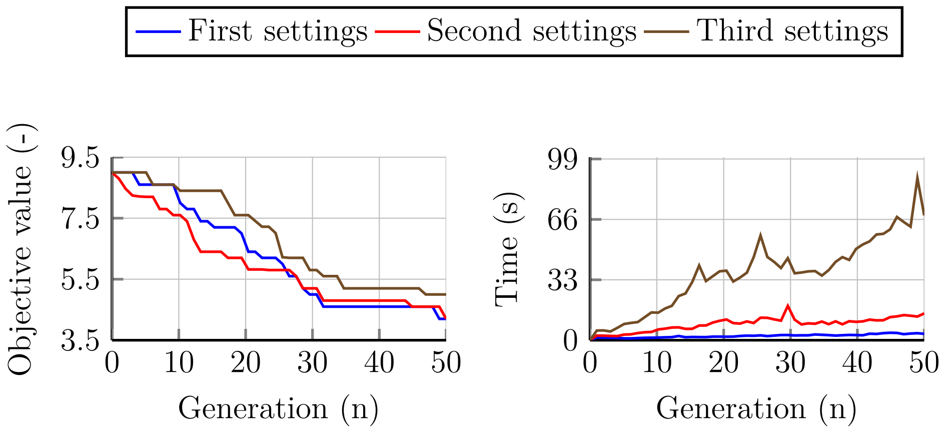

Figure 10: Time comparison of three different settings

It is clear from Figure 10 that the best possible choice for gaining the best result is the first or the second setting. The second setting has longer run-time. Clearly the first setting has the best results in this experiment. The first setting is used in the next section for further exploration.

It is unexpected that the solution with the highest time demands does not search for the solution more efficiently than the other solutions. It is also unexpected that the best solution has minimal time consumption.

[24]Without counting the errors gathered by predicate `shapeError/2`.

6.3.2 Concatenate run

The first setting described in the previous section is used for 20 runs of PRO-GP for 100 generations.

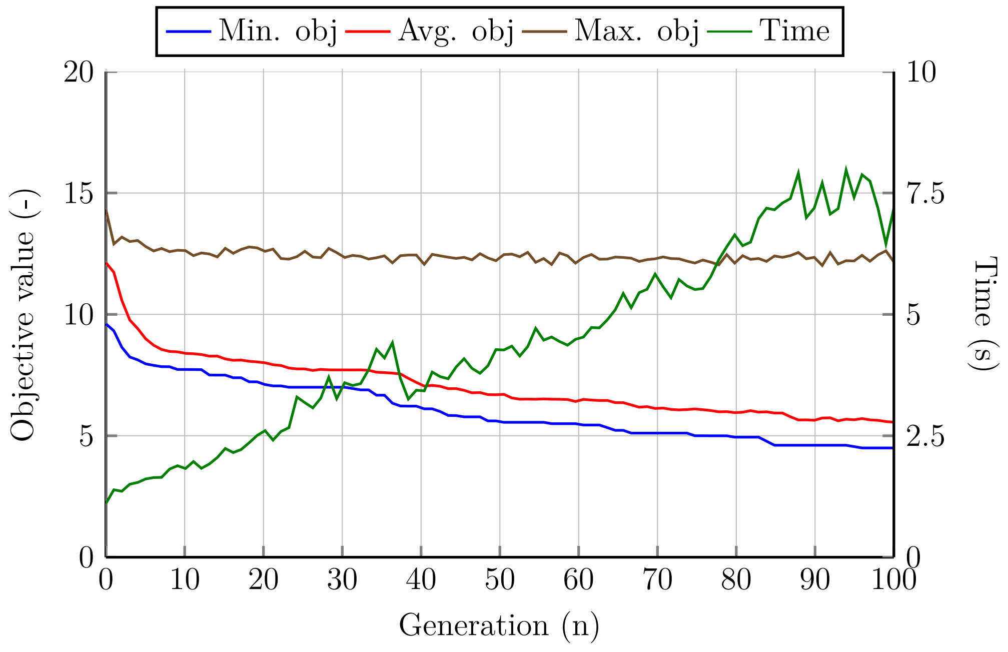

Figure 11: Concatenate

Figure 11 shows that PROGP does not have any significant loss in searching ability in this case; the average of minimal objective values decreases in the 100 generations. The time consumption is higher for every next generation but no generation exceeds the 10-second limit.

It is difficult to compare the results of the Concatenate task with the results from [Tang et. al. 1998]. The results of the genetic logic programming (GLP)[25] are not as informative as the results introduced here. There is no objective value improvement to compare. The only result that is comparable is the number of successful individuals and there are no successful individuals in either approach.

[25]The equivalent of PROGP implemented in Lisp and introduced in [Tang 1998].

7 Conclusion

The comparison of four BI algorithms (in section 2) results in finding two common principles: The first one is some kind of a fitness function. The fitness function as a general concept is used all over the AI. Its purpose is to say how well a certain task is solved in comparison with other solutions.

The second is an idea of evolution defined in the widest possible way, as described in [Dawkins 2006]. It is possible to evolve a genome (in the case of evolutionary algorithms), an ecosystem (in the case of ecologically inspired algorithms) or information[26] (in the case of artificial swarm intelligence algorithms).

The implementation of genetic programming in logical programming paradigm resulted in PROGP. PROGP is implemented in Prolog. It is based on PROGA [Olsson 1996]. PROGA has to be translated to SWI-Prolog. The translation was used for solving the knapsack problem. The source code of translated PROGA is included in the attachments (see attachment C). The source code of PROGP is very different from its inspiration. PROGP is also included in the attachments (see attachment C), which also contain three different fitness functions for three different tasks (as introduced in section 6).

A good property of PROGP is that it is not necessary to control whether an individual is a correct LP program. The definition of operators of reproduction does not allow PROGP to create an individual that is not an LP program. There is an exception in the source code of the fitness function that catches the error while loading the individual into the database. This exception did not appear while testing PROGP.

However, the PROGP implementation was not successful in solving even the simplest tasks. Possible causes for such behaviour are discussed in section 7.1 and in section 7.2.

The section 7.3 contains one of the possible solutions of one of the causes.

Some of the interesting conclusions include the time demands and recursion handling of Prolog. This is discussed in more detail in section 7.4.

Another interesting conclusion is the Prolog's errors Signal 11 and Signal 22. This is further discussed in section 7.5

7.1 Size of search space

The size of search space might be too large for using evolutionary programming algorithm. The search space depends largely on the variable types

[26]See the meme theory introduced in [Dawkins 2006]

used in the solution searching. For example, when searching for one variable of the type integer, it is necessary to search only for size:

$$10 + NrVar^{27}$$

In the case of the List variable, the size of the search space is much larger. at the beginning, there might be an integer or variable just like in the previous case. However, this can be replaced by a singleton or a list with two members. On every position of such lists, there can be an integer variable, a singleton, or a list with two members again. The maximum size of such list is set in PROGP to 2[28]. This limitation describes how deep every list type variable in the genome might be.

To search for one list variable means to search a space of size:

$$143 + 2 \times NrVar.$$

Consider the First Member task. There is the list of variables with two lists of variable types.

Searching for a set of variables means searching a space of size:

$$(143 + 2 \times NrVar)^2 = ListVar$$

To search for one predicate of the LP program means search in space of:

$$(NrPred + 1) \times ListVar$$

To search for a Horn clause of the LP program means search in a space of:

$$((NrPred + 1) \times ListVar + (NrPred + 1)^2 \times ListVar).$$

The first addend refers to the head of the clause. The second addend refers to the body of the clause.

The LP program might have several lines. The number of lines is limited only in the first generation. The general formula for the First Member task as defined in section 6.1 is as follows:

$$(MaxDepth \times ((NrPred + 1) \times ListVar + (NrPred + 1)^2 \times ListVar)$$

[27]During the creation of such variable it is possible to create it like an integer from 0 to 10 or it might be a variable.

[28]It is possible to increase the size. However, the size of the search space increases exponentially.

After substitution for variables:

$$6 \times (2 \times (2 \times (143 + 2 \times 5)) + 2^2 \times (2 \times (143 + 2 \times 5))) = 11\,016$$

The whole space would be searched by some of the deterministic search in the 55th generation.

It is mentioned in the results of the previous section that half of all runs found the result for this task in the 8th generation (with population size 200).

This result is not exact. In fact, it is a bit confusing. There is more than one successful result in the search space defined by the First Member task. For the Member task, the space is even smaller (there are only integer variable type and list variable type in the list of variables), but the searching algorithm does not reach the correct answer. The task is more difficult because there are fewer successful results in the search space. However, it seems that the search space is not the biggest problem of PROGP.

7.2 Why is PROGP unsuccessful?

Length control was suggested in [Tang 1998] as a possible improvement of their algorithm. PROGP applies length control. However, PROGP does not have significantly better results than the implementation introduced in [Tang 1998]. For more expressive results it would be necessary to apply more experiments and some further testing on PROGP and on the implementation from [Tang et. al. 1998].

There are several other possible explanations:

One of the reasons might be the insufficient size of I/O sets used in section 6. There is a relation between the size of an I/O set and the answer searching [Tang 1998]. This might be the reason why PROGP is unsuccessful.

The I/O sets were very limited due the PROGP time consumption. There was not observed much improvement with the change of the I/O sets sizes during the testing of PROGP. The 'quality' of I/O set was much more depending. The results are not statistically proved and it is possible to continue with the work in this respect.

Another reason might be the inefficiency of operators of reproduction. The operators were taken from usual GP definition. While testing PROGP, the operator of crossover was extended to include the predicate `xoverb/6` because the crossover definition was not successful enough during testing. The new definition had a positive effect on the evolution process. However, the effect was not positive enough.

Even the definition of mutation was changed during PROGP testing. The definition of mutation of predicates is the same as at the beginning.

The change came with the definition of mutation of a set of variables. Originally the whole set of variables was replaced with a different one. The second version changed one of the variables for a different one. The current version changes even the first level of the list variable (for details see section 2.2.1).

It is possible to think of different definition of operators of reproduction or it is possible to improve the impact of crossover and mutation by some sort of local search algorithm (for example by trying several possible crossover points and choosing the best one). According to the results, the crossover operator is too general and searches the space too widely (since most of the successful individuals were created by mutation).

PROGP might be very parameter depending. It is possible to evolve parameters via some ecological algorithm. For example, an expansion of PROGP on the PS2O algorithm is suggested in section 7.3.

7.3 PS2O application

The run of PROGP is probably very parameter depending (there is no proof of this claim). To make PROGP more general, it is possible to extend it on PS2O algorithm. Different species would have different settings of properties.

Parameters from the second group (recall section 5.3.1) might create species and help find the best possible parameters setting for a task. This approach would even allow the usage of the same definition of fitness function (that is necessary for PS2O).

PS2O algorithm might actually approximate the answer to the issue of parameter dependence. If the solution of PS2O algorithm was similar for every task, it would lead to the contrary of the claim from the beginning of this section.

7.4 Prolog recursion handling

The time demands of PROGP are a very interesting result on the level of Prolog (recall the result from section 6.2). The most natural experiment for testing the time consumption would be to rewrite PROGP non-recursively and consequently compare the time consumption of such program and PROGP.

This result is even more interesting because according to the on screen written statistics there is no working memory usage that would increase while PROGP runs. The only value that is increasing is the gained garbage collection. The duration of gaining the garbage collection is also increasing. However, while creating the next population took 2 seconds, gaining the garbage collection took approximately one tenth of a second. This result deserves future work.

7.5 Prolog failure and signal errors

Errors Signal 22 and Signal 11 occurred in the process of programming PROGP. These errors have something in common. The signal 11 did not appear after putting the predicate `retract/1` into the `catch/3`[29] predicate.

The official documentation of SWI-Prolog does not include these errors. There might be several causes of these Signal errors. The appearance of Signal 11 dropped after a roof of the length of the variable type list in the list of variables was set. That would suggest that the cause was a memory problem. Signal 22 is very probably caused by predicate `retract/1` which takes care of the database handling. Some further work in this respect is definitely possible.

7.6 Epilogue

Our implementation of genetic programming was based on the translation of PROGA into the SWI-Prolog language. This translation was successful and it solves the knapsack problem in few generations with adequate time demands.

However, PROGP application was not a successful implementation of GP for logical programming. This is not because the problem is too difficult. There are several ways how to try to improve PROGP: finding the best possible setting for a task, finding the best possible I/O set for a task, or trying different operators of reproduction.

To compare PROGP with other GP programming, it would be necessary to extend the genome language to the language of Prolog. The genome of PROGP can only be used in the language of logical programming. The language of Prolog contains operator of cut, operator of negation, and the language of arithmetic. The extension of language of the genome would increase the searching space and therefore probably lower the space searching capability of PROGP.

[29]The error handling Prolog predicate

A List of attachments

A List of attachments

B Translation algorithm (see section 4.1.2).

C Description of the attached files content

B Translation algorithm

The following algorithm translates any program in the language of logical programming into the restricted language (there are only predicates with the same arity).

1. Rename predicates with the same name but different arity.
2. Go through all predicates and make a list of couples. On the first position of the tuple is the name of the predicate and on the second position of the tuple is the sum of all arities that were gone through. Every predicate is in the list only once. Store it as PredicateList.
3. Add the last predicate arity to the number assigned to the last predicate. Store the number as NrVar.
4. Go through the program and for every predicate:
   (a) Find the number assigned to it in the PredicateList.
   (b) Substitute the predicate set of variables with set of variables starting with anonymous variables until the found number is reached.
   (c) Add the original variables.
   (d) Fill the variables list with anonymous variables until NrVar is reached .

Is such a program really correct translation?

Without loss of generality fix a LP program and call it original. Apply Translation algorithm on the original program and call the resulting program an extension.

The original program and its extension have clearly the same shape of the graph (as introduced in section 4.1.1). The program and its extension do not solve the same questions!

However, now it is simple to finish the translation:

To make a program solving the same questions, one would need to add at the beginning (or at the end) of the database a translation between the original and extended program.

To create such translation, recall the PredicateList used in the Translation algorithm. Each member of the PredicateList creates one new predicate. Head is the name of the predicate in the original program. Body is its equivalent in the extension with extended variables list just like in point 3) in the Translation algorithm.

This way for every input, the extension output is true only and only if the original program output is true.

C Description of the attached files content

There are two folderes on the attached CD-ROM (or in the downloadable 'koza_and_prolog.zip' file). The first is called PROGA. It contains the source code of PROGA translated into the language of SWI-Prolog.

The second is called PROGP and contains three subfolders. Each of these subfolders contains PROGP set for solving one of three tasks as described in the section 6

The folder tree of the attached files follows:

```
PROGA\
    fitness.pl
    ga.pl
    initialisation.pl
    lamarckian.pl
    misc.pl
    parameters.txt
    readinput.pl
    reproduce.pl
PROGP\
    concat\
      fitness.pl
      GP.pl
    first_member\
      fitness.pl
      GP.pl
    member\
      fitness.pl
      GP.pl
```

References

[Nilsson 2000] Nilsson, U., Maluszynski, J. (2000). *Logic, programming, and Prolog* (2nd ed., p. 276).

[Koza 1992] Koza, J. (1992). *Genetic programming: On the programming of computers by means of natural selection.* Cambridge, Mass.: MIT Press.

[Tang et. al. 1998] Tang, L. R., Califf, M. E., Mooney, R. J. (1998). *An experimental comparison of genetic programming and inductive logic programming on learning recursive list functions.* In 8th Int. Workshop on Inductive Logic Programming.

[Olsson 1996] Olsson, B. (1996). *PROGA - PROlog implementation of simple Genetic Algorithms.*

[Binitha et. al. 2012] Binitha, S., Sathya, S. S. (2012). *A survey of bio inspired optimization algorithms.* International Journal of Soft Computing and Engineering, 2(2), 137-151.

[Dawkins 2006] Dawkins, R. (2006). *The selfish gene (30th anniversary ed.).* Oxford: Oxford University Press.

[Back et. al. 1997] Back, T., Hammel, U., Schwefel, H. P. (1997). *Evolutionary computation: Comments on the history and current state.* Evolutionary computation, IEEE Transactions on, 1(1), 3-17.

[Wielemaker 2014] Wielemaker, J. (2014). *SWI Prolog Reference Manual 7.1 (1. Aufl. ed.).* Norderstedt: Books on Demand.

[Odum 1977] Odum, E. (1977). *Uvod: Napln ekologie. In Zaklady ekologie (3. vyd. ed., p. 733).* Praha: Academia.

[Britannica 2014] Multiple articles (see details in the actual citation). (2014). *Encyclopaedia Britannica.* Retrieved from *www.britannica.com.*