Gene Fragment Programming
Master thesis

Piotr Holubowicz

October 17, 2008

Promotor: dr hab. inż. Krzysztof Krawiec

Poznań University of Technology, Faculty of Management and Computing, Institute of Computer Science

Poznań, 2008 r.

Contents

1 Introduction

Genetic programming (GP) is an evolutionary computation technique that finds computer programs that solve user-defined problems [17]. Computer programs are treated as individuals that participate in a simulated evolution process, in which populations of programs interact with each other, and by means of natural selection only those that solve the problem best can prevail. Genetic programming is a meta-heuristic, which means that it does not attempt to solve any problem in a precise, deterministic way. Even more, the principles of GP stay the same independently of the sole nature of the problem it is used for; it is a general scheme of solving problems, in which only some stages need to be adjusted according to the particular application. Though GP can never surpass dedicated deterministic algorithms in terms of solution quality, it can surpass them greatly in terms of overall effectiveness, since it may reach satisfactory solutions in reasonable time even for many highly complex problems. For this reason GP has been successfully applied to a wide range of optimization problems, such as quantum computing, electronic design, searching and many more [5].

Genetic programming has become particularly popular since the publication of a book dedicated to GP written by John Koza in 1992 [14]. Since then a lot of effort has been made to enhance the general GP idea, by adding new features, trying to improve its weak points or adjusting it to more limited fields of applications. This work is an example of such approach, which is to design and examine a new methodology based on genetic programming, called the *Gene Fragment Programming* (GFP). The main motivation behind GFP is to develop a general framework for solving modular problems. To achieve this, GFP presents a novel approach in the field of genetic programming, which is to evolve fragments of solutions independently and subsequently link them in order to create a whole solution. Such approach may offer new possibilities in its field of application but also introduces a set of challenges that need to be addressed at the design and implementation stages. This work presents the process of developing gene fragment programming, analysis of its behaviour on a selected group of problems and the conclusions than can be drawn as their result.

2 Scope and objectives

2.1 Objectives

The **primary goal** of this work is to develop a new approach to genetic programming (GP) and examine it with the use of a selected set of benchmarks. Genetic programming is a methodology based on evolutionary computation (EC) that attempts to solve complex problems by generating computer programs in an evolutionary way. The new variant of genetic programming that is the topic of this work will be called *gene fragment programming* (GFP).

Apart from designing the algorithm, certain hypotheses are formulated concerning its features and the types of tasks it should be best suited for. In the experimental part of work those hypothesis will be examined and experimentally verified, and conclusions will be drawn.

The **specific objectives** of this thesis are as follows:

1. To present a novel approach to genetic programming that utilizes building solutions from small blocks, called gene fragments, thereafter referred to as *gene fragment programming* (GFP). In particular:
    (a) To review the existing variants of genetic programming that present approaches similar to GFP
    (b) To present the GFP algorithm, discuss design alternatives and motivate the choices made
    (c) To present hypothesis concerning GFP and a plan of their verification
2. To verify the proposed method on real-world tasks, in particular:
    (a) To assess the overall performance of GFP
    (b) To propose problems that should be particularly suited for GFP and verify its performance for them
    (c) To compare GFP to classic GP

2.2 Scope

A fundamental part of work was the process of designing the GFP algorithm itself. First, similar existing approaches were examined in order to determine which concepts had already been researched and with what results. Based on that, a general design of the GFP was developed. When designing the details, various options and parameters appeared. Some of them were resolved at the design stage, while others were left open and were researched experimentally at a later stage. As a result of that process, a structured description of GFP was formed together with a list of experiments to follow and hypothesis to verify.

Along with the theoretical work, implementation was carried out as well. First of all, implementing the algorithm worked as a 'proof of concept', which means it supported the design process by revealing the parts that were unclear or inconsistent and by verifying quickly some design alternatives. Furthermore, it allowed for extensive testing of the proposed approach and verifying the above mentioned hypothesis in practice.

Finally, as the experiment results were collected, they could be analyzed and discussed so that final conclusions could be made.

This document describes the above work and is structured as follows. Chapter 3 presents the approaches that form the basis of gene fragment programming and lists the technical terms that are used in the following chapters. Chapter 4 is dedicated to the theoretical aspect of GFP, describing and justifying in detail its design. This is followed in chapter 5 by a description of a few more specific methods that, though not directly related to GFP, are worth noting for being similar to it in some way. Chapter 6 is related to the implementation of GFP. Before running the experiments, the problems needed to be selected that would be most appropriate, and that is described in chapter 7.The experiments and the collected results are presented in chapter 8. Finally, chapter 9 contains the discussion of results and presents conclusions, summarizing the whole work. Bibliography is located at the end of this document.

3 State of the art

3.1 Evolutionary algorithms

Evolutionary algorithms (EA) [11] are meta-heuristic optimization techniques used in computing that are inspired by biological evolution. Potential solutions play the role of individuals in a population and are optimized by means of evolutionary mechanisms - reproduction, mutation, recombination and selection. Similarly to biological evolution, evolutionary algorithms do not make any specific assumptions about the nature of the problem, which rendered them useful in a wide range of problems.

The general schema of evolutionary algorithms may be outlined as follows:

1. Initialize a pool of potential solutions, called *population*. Each element of the population is called an *individual*.
2. Evaluate each solution and calculate its *fitness*.
3. Apply genetic operators:
   (a) choose some solutions and *recombine* them creating new ones
   (b) choose some solutions and *mutate* them, also creating new ones
   (c) *select* solutions that will survive and replace the rest with the new ones
4. If a stopping condition is not met, go to point 2.
5. The result is the best solution found, with respect to fitness.

Since genetic operators usually operate on a high level of abstraction, the part of evolutionary algorithms that is most strictly connected to the problem is the fitness function. It should evaluate solutions assigning high values to those that are close to optimal and low values to those far from it. Furthermore, it should be quick to calculate, as it is one of the basic steps in evolution and is called numerous times.

Closely related to the fitness function is the selection operator. It determines which solutions will be abandoned and will make place for new alternatives. Usually, this depends on their fitness value but is non-deterministic. Examples of selection operators include roulette, tournament and ranking selections.

The schema of evolution presented above is called generational, since populations are clearly separated from each other; a new population evolves only to replace the previous one (steps 2 and 3). This is the traditional model of evolutionary algorithms but not the only one. In steady-state algorithms new individuals are introduced to the population immediately, replacing a randomly selected 'poor' individual (the so-called 'reverse selection'). It is in general simple to implement and shows good convergence [21]. For convenience, the concept of generation is also used in steady-state algorithms meaning a group of steps that corresponds to a generation in generational algorithms.

Depending on the type of optimization problem and the form of solution that is accepts, evolutionary techniques can be further divided. *Genetic algorithms* (GA) are the most popular evolutionary algorithms, in which solutions are usually strings of numbers of fixed length that represent solutions of optimization problems. The two most commonly used operators are crossover and mutation. Selection methods include fitness-proportional, in which each individual is assigned probability of being selected proportional to its fitness, ranking, in which probability depends on the order of individuals and tournament selection, in which only a subset of population competes to be selected.

In *genetic programming* (GP) [14] solutions have the form of small computer programs that are supposed to perform a specified task. Traditional GP programs are represented as tree-structures and make use of two genetic operators: crossover and mutation. Genetic programming is the basis of gene fragment programing and will be discussed in detail in the following chapter.

3.2 Genetic Programming

Genetic programming (GP) can be defined as evolutionary computing that aspires "to induce a population of computer programs that improve automatically as they experience the data on which they are trained" [21]. Genetic operators were used for evolving computer programs already in the 1980's but it wasn't until the publication of John Koza's treatise in 1992 [14] that it became well-known in its current form.

The fundamental feature of GP that distinguishes it from other evolutionary algorithms is that it evolves *computer programs.* They can have various forms - most commonly they are tree structures, as proposed by Koza, in some variants of GP they are linear or graph structures or, more historically, PROLOG programs. In further discussion of GP it will be assumed that tree structures are used, unless noted otherwise. Regardless of structure, computer programs consist of two types of primitives, functions and terminals. Terminals represent inputs to the system or constant values while functions process internal, already calculated values. Terminal and function sets need to be defined before running the evolution. Programs are run for a defined set of inputs and fitness is assigned based on their outputs. Usually the ideal output is known a priori and fitness is calculated by comparing the program output with the ideal one but this is not a rule.

A strong point of genetic programming is its ability to evolve programs of different forms, represented by any valid combination of symbols from the alphabet. In practice, some constraints are used in order to limit the search space and increase efficiency of the algorithm. Koza puts a limit of maximum tree depth of 17, transformations that violate that rule are considered invalid. Certain approaches exist that try to define the general shape of evaluated solutions. One of them is the idea of Automatically Defined Functions (ADFs), introduced by Koza. ADFs are branches of tree with a determined input and output, similarly to functions in C or Pascal programs. ADFs evolve in parallel to the main tree and can be used many times. Alternative approach is to define GP individuals as a set of trees (forests) rather than single trees. Such sets can be evaluated in various manners, for example [10] defines in advance operators that join them and are not subject to evolution.

Genetic operators Genetic operators are the means to transform individuals in the process of evolution. Two principal operators used in GP are crossover and mutation.

- *Crossover* is meant to help evolution converge to optimum. Two individuals are selected, usually based on their fitness. Subtrees are then selected in each of them and swapped, if they do not break the tree constraints, such as depth or size. The goal is to support producing individuals combining good points of both parent individuals.

- *Mutation* introduces diversion to population. For a selected individual, it replaces a randomly chosen subtree with a new one. Similarly to crossover, mutation is only performed if the resulting tree does not violate the tree constraints.

Other operators can also be used, such as *permutation*. Permutation is similar to mutation, only that instead of exchanging the subtrees, it swaps just a pair of nodes that are structurally equivalent [10].

Fitness and selection Fitness is a measure of quality of a candidate solution. It guides the evolution by introducing *selective pressure* that tells apart the well-performing individuals from the inferior ones. Individuals with higher fitness have bigger chances of being the source of new individuals whereas lower fitness increases the probability of being replaced with a new individual. It is important that the fitness function is continuous, that is that small improvements in quality of an individual should be represented by small improvements of its fitness, while big improvements in quality of an individual should be represented by big improvements in its fitness [21].

Fitness function is related strictly to problem domain. In GP it is calculated by running the evaluated program on a training set of data and assessing the output of the program. For some problems the ideal output is known a priori and fitness can be calculated by comparing the program output with the ideal one, which often includes running on a multiple set of data. In other cases, such as the artificial ant problem, the resulting program is run on in a certain environment, where it performs a set of operations and the final result is taken into consideration.

Traditionally, by analogy with biological evolution, fitness is maximized. Many fitness functions exist (some are minimized and are typically further processed later), for instance:

- The squared error, for symbolic regression problems ($o_i$is the correct i-th output, p_i is the i-th output of the evaluated program)
$$f = \sum_{i=1}^{n} \left(p_i - o_i\right)^2$$
- The number of correctly classified examples, for classification problems
- The amount of food found and eaten by an artificial agent in an artificial life application [21]

In many cases, the above values are further normalized, that means transformed so that the fitness is always between zero and one. Examples include scaling program outputs by the ideal output (if it is known) or converting fitness f using the formula $f' = \frac{1}{1+f}$, into the scale from 0.0 exclusive (worst) to 1.0 inclusive (best)[1].

In order to use genetic operators, a selection method is necessary that would pick up individuals from the population, based on their fitness. In general, when looking for 'good' candidate solutions, individuals with

[1]This method is used in ECJ, called 'adjusted fitness', see chapter XXX

higher fitness should have bigger chances of being selected, and vice versa. The basic selection method is *roulette*, in which every individual i is given a probability of being selected of

$$p_i = f_i / \sum_j f_j$$

This method, however, does not work well in situations where individuals have very similar fitnesses, since the probability depends on absolute fitness rather than on differences between individuals. This issue is addressed by tournament selection. In this method, a set of individuals is randomly chosen from the population and their fitnesses are compared; the best one is selected. This procedure is repeated until the desired number of individuals are selected. The size of the tournament varies, the common sizes are 2, 5 and 7 - the higher the bigger the selection pressure.

A feature of selection that can be applied regardless of the selection method is *elitism*. A set of best individuals (typically only one) is saved at the beginning of every generation. Should the best individuals of the next generation have worse fitnesses than the saved ones, the latter are introduced back to the population and in this way the information about the best individuals so far is preserved.

Benchmarks There is a variety of problems that genetic programming algorithms can use as benchmarks. The most common one is *symbolic regression*. The objective is to discover a mathematical expression given a set of data that correspond to it. It is called 'symbolic' because the task is to discover the model of the expression, not just the coefficients. For a function of one variable, the input for the problem is a set of randomly selected points. The fitness is measured as squared error for each of the points. Additionally, Koza defines a 'hits' measure, which is the number of input cases (points) that are solved correctly - that is, the program output fits within a small error rate the correct value. Examples of symbolic regression include:

- $y = x^4 + x^3 + x^2 + x$ (called 'quartic')
- $y = x^5 - 2x^3 + x$ (called 'quintic')
- $y = x^6 - 2x^4 + x^2$ (called 'sextic')

In the *ant trail* problem, the task is to direct an ant moving on a virtual plane so that it collects maximum number of pieces of food located on it. The most well-known version is called the Santa Fe ant trail, in which 89 pieces of food are located on a 32 by 32 toroidal grid (top and bottom are connected, as well as left and right sides). The ant can turn left, right, move one square forward and may also look ahead one square in the direction it is facing, to determine if that square contains a piece of food [1]. The quality of the program is measured by the time (number of steps) in which it accomplishes the goal.

Multiplexer is an example of a boolean problem. In such problems, input cases are boolean - they can only be passed or failed. A multiplexer is a device that reads the address of an input line and forwards the value of the line to the output. The objective of this problem is to develop a program that does exactly the same thing. In the most common variant, the 11-multiplexer, the program is given 3 lines that contain the address of one of the other 8 lines. Programs can be evaluated with the full set of $2^{11} = 2048$ possible inputs. The fitness is calculated based on the number of cases solved correctly.

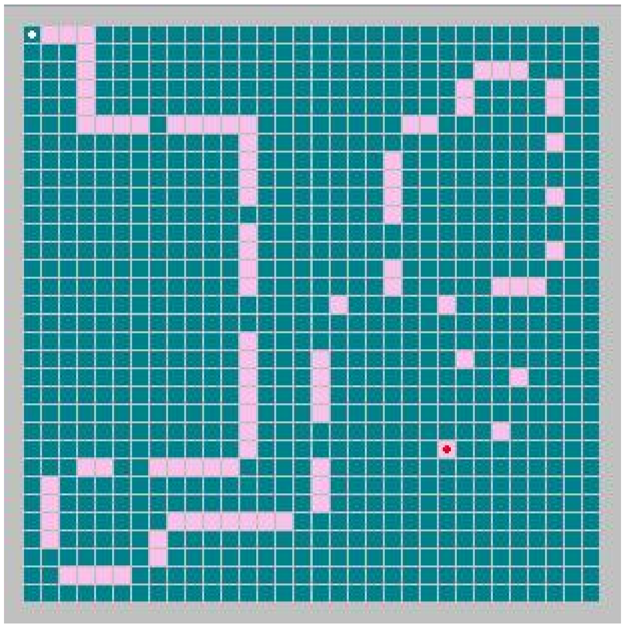


Figure 1: The Santa Fe ant trail [6]

In *parity* problem the task is to classify sequences consisting of 1's and 0's according to whether the number of 1's is even or odd. The GP problem can use terminal symbols related to each element of the sequence and use logical gates (such as AND, OR and NOT) to construct an expression returning 1 or 0 on even or odd number of 1's, depending on the configuration. For example, an even-31 version, means that the given sequence consists of 31 digits and the program should return 1 if the number of 1's is even, and 0 otherwise. Parity problem has been shown to be very difficult [13].

3.3 Artificial Chemistry

Evolutionary computing aims at finding solutions of optimization problems by taking inspiration from the real-life process of evolution. Possible solutions are treated as individuals of a population that by being subject to rules of evolution evolve towards the optimum. What is common for these algorithms is that they take solutions as entities - that is, at no moment they are constructed from smaller particles nor they function as elements of a bigger structure. Even though evolutionary algorithms need many individuals to run, they need only one to solve the problem - it is an individual that is the result, not the population. Among optimization algorithms that do not have that feature is *artificial chemistry* (AC). It is a subfield of *artificial life* that shares the inspiration from Darwin's theory of evolution but from a much different, low-level point of view. Artificial chemistry is based on the concept of *emergence*, which is the ability of small particles to create complex systems with the use of simple rules of interaction [16].

It deals with artificial chemical systems in which a set of molecules S is dynamically changing according to rules of reactions R and dynamics of the system A. The form of molecules depends on the type of AC, it can be abstract symbols, character sequences, numbers or tree structures, among others. Rules of reaction describe what happens when two or more molecules interact. In general they have the following form:

$$s_1 + s_2 + ... + s_n \rightarrow s_1^{'} + s_2^{'} + ... + s_m^{'}$$

Reaction rules can include additional parameters such as neighbourhood, probabilities or energy consumption. Dynamics of the system represent the environment for the reactions. First of all they define how the set of molecules is represented in the system - most typically it is either a multi-set or a concentration vector (when all molecules are defined a priori, and what changes is their amount in the environment), it can also incorporate a spatial structure of some kind. Secondly, there are different approaches to applying the reaction rules themselves. In the basic approach, random particles are repeatedly drawn from the multi-set and a reactions occur, whenever possible. When concentration vectors are used, it is more common to treat the rules of reaction as equations describing the changes in the concentration, and to run the simulation by their iterative application. Overall, [16] names 3 basic group of applications for AC:

- modeling - AC can be used to simulate real-life distributed systems, both on the micro- and macro scale
- information processing - artificial chemical systems combined with domain knowledge can give insight into processes happening on the 'chemical' level - movement of bacteria, brain functioning and others
- optimization - AC can be treated as a form of evolutionary computing and used for solving optimization problems

As far as gene fragment programming is concerned, the idea of a set of molecules - a so-called *soup* - is the most valuable concept of artificial chemistry. In fact, similarity between the soup of molecules in AC and population of individuals in GP is one of the main reasons for treating such evolutionary computing systems as representants of artificial chemistry. The idea of GFP is based on the belief that good solutions can be constructed from simple ingredients 'floating' freely in the environment. It has to be noted, however, that the notion of 'solution' is different in AC than in genetic algorithms and the rules of interaction are also defined in a significantly different manner. Gene fragment programming does not try to combine all of them in one approach, it rather takes inspiration from some concepts of artificial chemistry and applies them to the classic approaches of genetic programming.

4 The gene fragment programming approach

4.1 Overview of the algorithm

In genetic programming, the theory behind the process of creating successful solutions is still under development. A widely known theory, the building block hypothesis (BBH) [11], states that the success can be due to the fact that during evolution groups of genes are formed and propagated that have an above average fitness. Such groups, called building blocks, would be then implicitly used as parts of individuals. The gene fragment programming (GFP) is based on classic genetic programming but is different in that it attempts to explicitly express the functional fragments that create the solutions. The main concept of GFP is to *evolve directly small pieces of computer programs that are subsequently assembled to a final solution.* That means that evolution happens at a level lower than typically - it operates on modules of solutions and connections between them, while solutions are only generated on that basis. There are two types of benefits that may arise from such approach:

- *support for the emergence of modularity in GP* - in evolutionary computing, two most commonly used genetic operators are mutation and crossover (recombination). The former introduces diversity to population, since it exchanges parts of individuals with new, randomly generated equivalents. The latter helps the population converge in the optimum by combining two individuals in order to generate a new one that can share their good points. However, in GP the crossover operator is widely criticized for being rather destructive as there is practically no constraints on the selection of parts to exchange; any building blocks that may be formed within solutions can be easily broken. In GFP, where building blocks are evolved separately, the chance of their preservation should be higher since a useful building block will be directly assigned high fitness and will not be deleted from the population. With this feature GFP supports emergence of functional modules during evolution.
- *automatic problem decomposition* - when solving problems it is often useful to be able to decompose them to simpler sub-problems and then solve each of them independently. It is often very difficult to do that manually either because the problems are too complex or the knowledge about their nature is not sufficient. With the support for modularity, GFP should be able to implicitly decompose problems automatically. This can happen because fragments are evolving independently as individuals and positive evolutions of some of them can be awarded regardless of transformations applied to others.

The flowchart of GFP is presented in figure 2. First, initial population is created. Then, iteratively, a solution is assembled from fragments and evaluated. Fitnesses of fragments are modified according to fitness of solution they created. Finally, genetic operators are applied to keep the evolution going. These steps are repeated until an ideal solution is found or a maximum number of generations is reached.

The scheme presented in figure 2 demonstrates the high-level concept of gene fragment programming and hides many issues that need to be solved in order to make GFP work. The most basic one is the structure of gene fragments and the mechanism of linking them when creating solutions. Next, a method of selecting fragments that will create the solution needs to be defined. Finally, once a solution is created, fitnesses of fragments that constructed it should be modified accordingly. The following chapters discuss various approach to these topics.

4.2 Fragment structure

Gene fragments are individuals that are able to contain functional blocks and can be linked to one another in order to create a complete solution. Solutions in GP are typically represented as tree structures and so it is in GFP. It is natural therefore to represent gene fragments as (sub)trees that can be linked by joining a leaf of one tree with the root of another. Such representation is advantageous in that it does not impose any

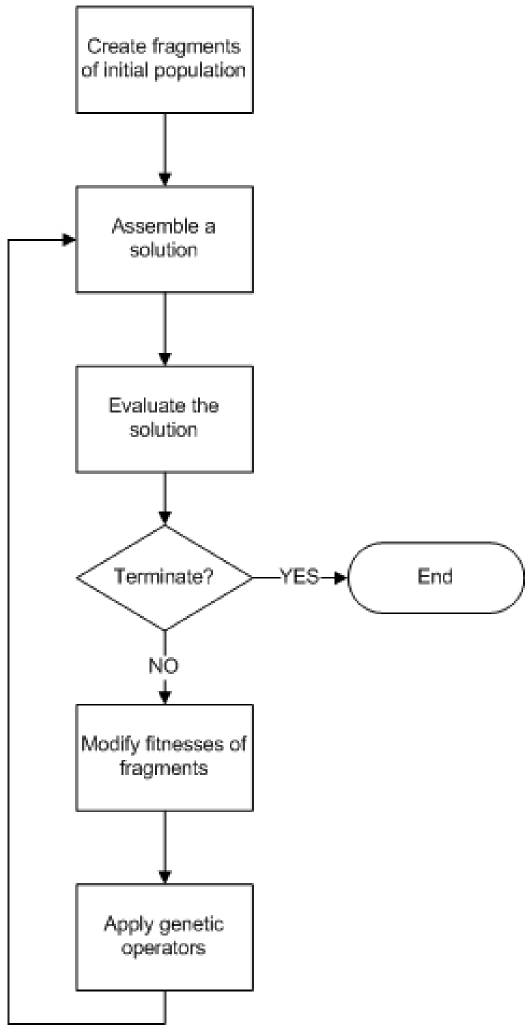


Figure 2: The general flowchart of GFP

additional constraints on fragments compared to their GP equivalents, neither in their shape nor in the set of nodes that can be used.

When two fragments are linked, the root node of one of them *replaces* the leaf node of the other. In this way, connecting any two fragments is always possible[2], regardless of node arities. The side effect of such connection is that the value of the leaf node is ignored. It should not be considered a drawback, however, taking into consideration that fragments are always evaluated as parts of solutions and their fitness is assigned regardless of the values of terminal nodes that were ignored because they participated in connections. In fact, an idea of a terminal symbol with no value assigned that could be used only for connections (a so-called *connector*) was also considered. It was abandoned because initial research showed that it puts strong restrictions on constructing solutions - not every combination of fragments was valid - without bringing any clear improvements.

4.3 Building solutions from fragments

Solutions are built from fragments, by selecting the first one and adding new ones iteratively until the solution is ready. The scheme is as follows:

1. Choose first fragment of solution.
2. Add fragment to solution.
3. Is solution ready? If yes, then stop.
4. Identify all possible connection points.
5. Find all candidate fragments that can be added.
6. Select one fragment from the candidate set.
7. Go to point 2.

The first fragment is generally close to the center of the hyperspace, it is searched for just as it is done in point 5, when looking for candidate fragments. The only difference is that at the beginning, the connecting point is assumed to be the center of the hyperspace, while later it is the set of all the leaf nodes of the solution (point 4). Connecting two fragments has already been described in chapter 4.2. However, there are still issues that need to be resolved:

- how to select candidate fragments from the population, both for step 1 and 5? This is addressed by the *fragment selection method.*
- how to choose one fragment from a candidate set? This is done by the *solution expansion method.*
- how to decide, whether the solution is ready or not? This is determined by the *build-stop method.*

Al three methods that are used for solution building are discussed in the following chapters.

[2]provided that it does violate not other rules of experiment, such as maximum tree size/depth.

4.3.1 Fragment selection method and spatial information

Requirements. Solution is built from fragments in an iterative manner, by adding new fragments one by one. Each such step consists first of identifying a set of fragments that should be considered as potentially worth adding, and secondly choosing one fragment from that set. The former sub-step is realized by the *fragment selection method.* Fragment selection method receives a current form of solution as input and is expected to return as set of candidate fragments that may be added (more precisely, it returns a set of pairs (f_i, n_i) where f_i is the candidate fragment and n_i is the node it should be connected to).

Additionally, fragment selection method should meet the following requirements:

1. It should be repetitive. In the same state of evolution, two runs should construct similar solutions with high degree of probability. In other words, the fragment selection method should not be random.
2. It should take into consideration which combinations proved promising in previous runs.

Fragment selection method can be deterministic but does not have to (as long as it is sufficiently repetitive).

Spatial information. In GFP, the fragment selection method is implemented using the spatial information about the nodes. Each node of every fragment is assigned a *location* in an n-dimensional hyperspace. The closer two nodes are located, the higher the chance of creating a connection between them. Locations are modified during evolution, so that good connections are getting more probable in further runs, and bad ones are getting less likely. For convenience, locations are kept normalized during evolution, so that the average distance of fragments to the center of the hyperspace ($v_0 = [0, 0, ..., 0]$) is equal to 1.

More formally, each node i is assigned a real-numbers vector v_i of a length of n. The distance between two nodes i and j is defined as

$$d_{ij} = \sqrt{\left(v_{i1} - v_{j1}\right)^2 + \left(v_{i2} - v_{j2}\right)^2 + ... + \left(v_{in} - v_{jn}\right)^2}$$

Vectors v are modified after every run of the evolution to keep the condition

$$\frac{\sum_{i=1}^{m} d_{0i}}{m} = 1$$

where m is the number of gene fragments, d_{oi} is the distance between the root of the i-th fragment and the central point and $v_0 = [0, 0, ..., 0]$.

What is important about spatial information is to notice that it applies to individual nodes, not entire fragments of solutions. That means that a fragment of solution can have its nodes spread all over the hyperspace. That is desired, since each node may have their own set of fragments that it is most likely to be connected to. In practice, it is the locations of root and leaf nodes that are most important during evolution, for only they can form connections with other fragments. Location is assigned to all nodes for simplicity, but it also turns useful in some situations - for example some genetic operators can create new individuals that have root nodes that used to be 'internal', and in this case their location becomes significant.

Spatial information vs. fitness. Spatial information is not an alternative to fitness assigned to gene fragments. Apart from being assigned to every node of a fragment, unlike fitness, it serves a much different function. Fitness is used to assess the *usefulness* of a fragment and as such decides which fragments will be replaced with new ones, and which should be taken more frequently for the breeding process. Spatial information, on the other hand, describes *how the fragment should be connected* with others and is used only during the solution construction process. Even though these measures are strongly related to each other, since modifications to spatial information are done on the basis of how well certain connections were evaluated, it is important to note that their domains and application are different.

Implementation alternatives. With the set of requirements defined above and the spatial information that is used for preserving information about good connections, there is more than one concept of implementing the fragment selection method. Below are described different options that have been considered when designing the algorithm. What they have in common is that they all give higher chance to fragments that are located closer to their respective nodes, which is the principle of using spatial information. They differ in the distribution of probability based on the distance, the average number of fragments they return and, most probably, in speed and usefulness.

- Closest - each node that is a potential point of connection (i.e. a leaf node of existing solution), a so-called *connection node*, adds exactly one candidate fragment whose root node is located closest
- Roulette - each connection node takes into consideration a set of fragments located closest to it and selects one with roulette selection based on distances
- N-closest - generalization of “closest”; each connection node adds exactly N closest fragments to the candidate set
- All-in-range - each node adds all fragments that are located not further than a given distance (range)

Additionally, a simple “random” method can also be used, in which every node adds one randomly selected fragment to the candidate set. This does not meet the requirements of the fragment selection method defined above, but will be used later for analyzing the efficiency of other fragment selection methods.

4.3.2 Solution expansion method

Once the fragment selection method returns a set of candidate fragments that can be added to the existing solution, one of them needs to be selected and applied. This is done by the solution expansion method. Since its goal is to select one fragment from a given set, it can be implemented using any standard selection method. In GFP two alternatives were investigated: roulette and tournament selection. Fragments are compared based on the distance to the nodes they would be connected to.

4.3.3 Build-stop method

In chapter 4.3 a general scheme of constructing solutions from fragments is presented. How to add new fragments to existing solution was already discussed above but there is still an important point missing - how to detect when the solution construction process is complete. Two general approaches can be considered in GFP - *local*, in which fragment selection method (or solution expansion method) decides that there is no good candidate to add to the solution, and *global*, in which the process is finished regardless of these methods, based on another type of condition.

Local method depends on the type of fragment selection method used. In the “closest” and “N-closest” variants the construction process could be finished if the closest fragments are considered to be located too far (i.e. outside a specified distance limit). Similarly, the “all-in-range” would stop the process if the number of candidate fragments is below a certain limit. The benefit of these methods is that the moment when the construction process is stopped is a part of the spatial information and can be modified during evolution. However, in order to make it work local methods require an additional stop parameter that may be hard to define in advance - such as the minimum number of fragments in the “all-in-range” approach. It is a particularly significant problem taking into consideration that there are no assumptions about the distribution of spatial information, that is, in some cases fragments might tend to be grouped closely and in others to be dispersed in the hyper-space.

Global methods can use more straightforward conditions to stop the solution construction process. These include maximum size, when the process is stopped when solution consists of N nodes, and maximum depth - the process is stopped when solution depth is equal to M (where N and M are values defined in advance). An advantage of these methods is that their parameters are relatively easy to define, since it can be usually assumed that solutions should not exceed certain size or depth. This means, however, that the construction process will always stop after generating a solution that meets the border condition, not after generating a solution that it considers complete. In other words, it is likely that the best solution would be generated before the solution construction process terminates, and the scheme in chapter 4.3 should be modified respectively.

The above classification does not complete the list of possible build-stop conditions. In particular, a greedy method was also considered, in which the construction process is stopped when the difference between the fitness of the new solution and the fitness of the old solutions is lower than a given limit. Though greedy approaches turn out to be successful for a range of problems, it is hard to expect that in case of solution construction, mainly because it can be easily shown that when adding a new fragment to a solution its fitness does not need to increase, even if the final solution is evaluated as good.

4.4 Evaluating solutions

4.4.1 Calculating fitness of fragments

After a solution is constructed from gene fragments, it is evaluated and fitnesses of fragments are modified. In general, the goal is to increase fitness of fragments that participated in well evaluated solutions, which may but does not have to involve decreasing fitnesses otherwise. There are two major issues related to the "award method":

- rate of increasing/decreasing fitnesses - fragments may participate in many solutions during evolution, whose evaluation may vary a lot. Award method can be greedy, by simply assigning new fitness based on the most recent evaluation, or inert, by only modifying current fitness according to recent evaluations. In the latter case, the award method may be more or less inert, or may depend the inertia on whether the modification is positive (increase of fitness) or negative (decrease).
- inactive fragments - while some fragments may be used frequently, others may not even be used once. The fitness assigned to them will influence the course of evolution: if on average it will be lower than an average fitness of a frequently used fragment, they will tend to be removed from the population. In the opposite case, they will tend to be used instead of fragments that were used frequently but with little success.

4 variants of the award method have been developed to address these issues.

- classic - the fitness of a solution is assigned to each of the fragments, replacing the old one.
- inertial - the new fitness of a fragment is a weighted average of the old fitness and the fitness of the current solution: $f_i' = pf_s + (1-p)f_i$, where f_i' is the new fitness value, f_s is the fitness of the whole solution, f_i is the old fitness value, and p is the weight assigned to the new fitness
- asymmetrical - similar to inertial, but the weight is changed depending on which of the two fitnesses, old and new one, is better: $f_i' = p'f_s + (1-p')f_i$, where $p' = p$ if $f_s > f_i$ and $p' = (1-p)$ otherwise.
- decay - similar to classic, but fragments that were not used in the current solution have their fitness decreased by a given rate.

4.4.2 Modifying locations of fragments

After a solution is evaluated, locations of fragments are modified to reflect that evaluation. The idea is to make the distances between the fragments the smaller, the better those connections are - i.e. the higher the fitness of the solution is. This may involve bringing the fragments closer or moving them away but in practice only the former is necessary provided that the whole hyper-space is kept normalized (so that it does not collapse to a point). Furthermore, for every connection both the leaf node and the root node could be moved, but in GFP only the root location is modified, for simplicity.

The relocation procedure takes the calculated fitness of the solution as input. In GFP fitness is normalized between 0.0, worst, and 1.0, best. This allows for a simple relocation method, in which the root node is moved closer to the leaf node by a percent equal to the overall solution fitness:

$$v_r' = v_r + (v_l - v_r)f_s$$

where v_r' - new root location, v_r - current root location, v_l - leaf location and f_s - normalized solution fitness.

Complementary to the relocation procedure is the space normalization procedure, that scales all locations so that they do not diminish to $\vec{0}$. Two approaches were investigated for this procedure:

- absolute scaling - locations are scaled so that the maximum absolute value for each dimension is 1.
- distance variation scaling - locations are scaled so that the average distance to the central point of the hyper-space is 1.

Initial research proved the second method to disperse the locations better and thus it was used throughout the experiment. It was applied after every solution construction process.

4.5 Operators

Gene fragment programming uses 2 genetic operators typical for genetic programming: crossover and mutation. They are applied to gene fragments, since it is fragments that play the role of individuals during evolution. Location of every node is moved with the node when the nodes are swapped and it is randomly generated when new nodes are created.

5 Related work

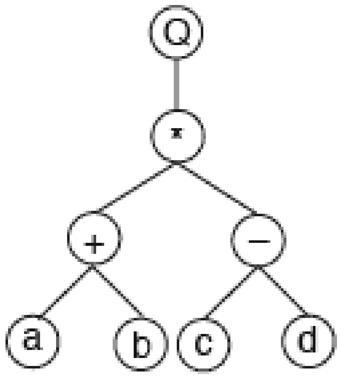
5.1 Gene Expression Programming

In genetic algorithms, individuals are linear strings of fixed length, while in genetic programming they are nonlinear entities with variable size and length, most commonly parse trees. Gene Expression Programming [10] combines those two approaches by adding genotype-phenotype mapping to traditional GP algorithms. Individuals (genotypes) are linear strings of fixed length which are later encoded to nonlinear entities. The main motivation of such feature is to combine the benefits of both representations: linear, fixed length strings are easier to manipulate genetically, while GP programs of variable size and shape may have big problem solving capabilities.

Each genotype in GEP may describe more than one parse tree, the part corresponding to each tree is called *a gene*. The parse tree - linear string coding procedure is relatively simple, as all nodes are encoded one by one according to their depth in the parse tree, starting with the root and finishing with the most right leaf node. An expression

$$\sqrt{(a-b)\cdot(c-d)}$$

can be represented as the following GP diagram:



where “Q” is the square root function. This is the phenotype in GEP and its corresponding genotype would therefore be: `Q*+-abcd`. The opposite procedure, which transforms linear genotype into a parse tree, is also easily doable knowing the arities of each node.

Clearly, not every combination of symbols represents a valid parse tree with respect to the above coding/encoding procedures. For this reason, each gene in the GEP genotype is divided into two parts, the head and the tail. The head can contain any symbols, both from the function set and terminals, whereas the tail contains only terminals. Lengths of the head h and tail t part are fixed and must meet the condition

$$t = h(n-1)+1$$

where n is the maximum possible node arity. Using fixed length chromosomes to encode variable size phenotypes leads to having some elements of the chromosome possibly unused (neutral). On the other hand, any modification to genotypes, which does not violate the the head and tail restrictions, results in a valid genotype.

The main characteristics of the GEP algorithm are the genotype structure and the genotype-phenotype mapping procedures. Potential advantages of genotype-phenotype mapping were discussed by Banzhaf in Binary Genetic Programming [8], introducing neutrality has also proved beneficial in some algorithms [12]. GEP in turn was successfully tested on a wide range of problems, including symbolic regression, Santa Fe Ant Trail, sequence induction and evolving cellular automata [10].

Gene Expression Programming is innovative in that it proposes a new way of evolving classic genetic programming solutions, which is similar to the main motivation behind the Gene Fragment Programming. Although the principles of GEP such as phenotype-genotype mapping remain much different to GFP, the positive influence of introducing neutrality is a feature that should hopefully be common to both approaches.

5.2 Cartesian Genetic Programming

Cartesian Genetic Programming [20], similarly to Gene Expression Programming, takes advantage of genotype-phenotype mapping. The encoded phenotypes are not typical GP parse trees, however, but have a form of directed graphs, in which nodes are addressed in a Cartesian coordinate system. Nodes are organized in layers, and each node can accept as inputs outputs from a given number of lower layers, including the problem data layer. The top layer contains one node that returns the overall result.

The structure of a program in terms of nodes layout is fixed, as it the number of inputs of each node. This leads to double redundancy - firstly, some nodes may not be used at all for calculating the ultimate result. Secondly, depending on the arity of the operator in each node, the number of inputs it uses may be smaller than the number of fixed connections to other nodes. Authors argue that, similarly to GEP and other algorithms, redundancy in genotype encoding is in fact with benefit to the effectiveness of the algorithm.

A set of parameters need to be defined before each run, most of which refer to the structure of the program. These include the nodes layout, the number of inputs and the inter-connectivity constraint that defines how many layers back can be connected to the current node. Given these parameters, a CGP program can be encoded with a simple set of integers, that define for each node the sources of its inputs and the code of operator that it represents. En example of a genotype/phenotype pair is shown on figure 3.

0 1 3 *0* 1 0 4 *1* 3 5 2 *2* 8 5 0 *1* 6 8 3 *2* 4 2 6 *0*
6 10 7 *2* 9 11 10 *1* 8 8 6 *2* 9 10 13 *15* 13 14 11 *16* 10 10 14 *2* 13 16 17

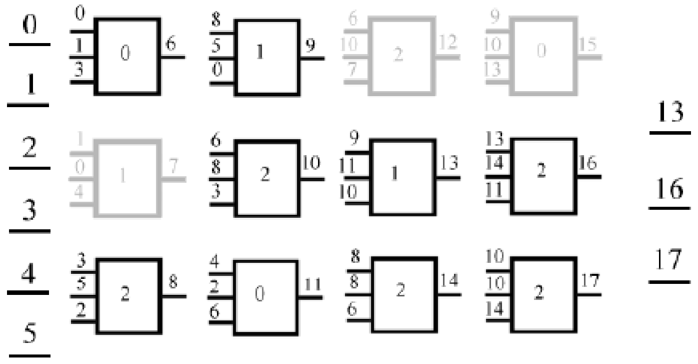


Figure 3: Genotype-phenotype mapping [20]: a) Phenotype. b) Genotype. For a program with six inputs and 3 outputs, and three functions (0, 1, 2 inside square nodes, in italics in genotype). The gray squares indicate unconnected nodes.

Genetic operators applied to CGP genotypes need to meet stricter requirements than in GEP, due to the more complex phenotype structure. For this reason, it often run with mutation operator only, without crossover [20]. The results showed that it solves Boolean problems particularly well but is also quite effective for the Santa Fe Ant Trail problem.

Cartesian Genetic Programming shares with GFP the concept of assigning coordinates to elements that comprise the solution, although in CGP they are merely used for defining the constraints on the data flow within an individual and are not subject to evolution.

6 Algorithm implementation

6.1 Software tools

6.1.1 Evolutionary Computation for Java

Gene Fragment Programming has been developed using ECJ, Evolutionary Computation for Java [2]. It is an open-source research environment written in Java designed to support running a wide range of EC problems, with significant configuration and modification possibilities. Its core is a group of highly abstract classes that are responsible for the key EC features, such as individuals, genetic operators, reproduction and evaluation. These are later subclassed by concrete classes that are dedicated to different fields of EC, among them genetic programming, evolutionary strategies and genetic algorithms. These classes are in turn connected to classes characteristic for each approach or problem, for genetic programming this includes GP parse trees and nodes, for example. Finally, the selection of classes to be used together with their parameters are put in a user configuration file, passed as a parameter when invoking the ECJ program.

With such modular structure, ECJ is highly customizable. Any element of the framework can be modified or exchanged, facilitating experimenting with enhancements to existing methods as well as implementing new ones. What is more, ECJ is flexible in terms of deployment - it is platform-independent, with support for multithreading and parallel computing.

GFP has been developed using version 17 of ECJ.

6.1.2 SVN and Trac

Subversion (SVN) [3] is a version control system, used to maintain current and historical versions of files such as source code, web pages and documentation. Though it is often used for teams that work together on one project, it is useful also for single users, mainly by maintaining historical versions of files thus allowing to revert any modifications or to maintain various branches of a project. During GFP development, SVN was used for two components - the software, developed with ECJ, and the documentation, consisting of documents, experiment results, bibliography and images.

To control the progress of work, the Trac system [4] was used. It is a web-based project management tool, written in Python and integrated with SVN. It was used mainly for its ability to register tasks associated with the project and automatically update their status according to the changes in documents committed to the SVN database.

6.2 Project design

GFP was implemented as a modification of a classic, Koza-style GP available in ECJ. Most changes were related to 2 aspects:

- solution construction - gene fragments had a structure much similar to typical parse trees except for that they did not represent a complete solution but needed to be linked in order to create one. This required modifications to classes representing leaf nodes for each problem (for example node “X” representing input in symbolic regression problem) as well as classes responsible for evaluating the response of a solution.

- assigning coordinates to nodes - nodes in GFP are located in multi-dimensional space; classes representing the nodes, trees and whole individuals needed to be adjusted to include this information in common actions such as cloning and producing new individuals

Additionally, new classes were created that were responsible for GFP functionalities. They were grouped in the following packages:

- ec.app.thesis - the main GFP package, contains core ECJ classes modified to enable new solution construction and evaluation process. This includes also the statistics class, used to generate most of the experiment results
- ec.app.thesis.builder - classes responsible for the solution construction process
- ec.app.thesis.spaceexplorer - classes related to spatial information, dealing with modifying the locations after each evaluation as well as resizing the whole space according to maintain the average distance to the central point constant
- ec.app.thesis.parity.func, ec.app.thesis.regression.func, ec.app.thesis.ant.func - contain modified classes representing leaf nodes of respective problems, necessary to include information about fragment connections during the evaluation process

Since many methods of GFP have a few alternative implementations, the *factory method* design pattern was used. That means that for fragment selection method, for example, a general method was always invoked which, according to the parameter supplied by the user in the configuration file, invoked in turn the correct implementation of that functionality.

6.3 Implementation challenges

When implementing GFP, the initial intention was to develop a platform as universal as possible, ideally in a form of a “wrapper” around the GP problems - that is, a framework in which any GP problem can easily be tested by simple changes in the configuration file. This turned to be a problem with respect to the novel approach of evolving solutions spread across more than one individual. In the end, the changes in code necessary to adjust it for a particular problem were limited to two places - the evaluation class, that had to be linked to the original problem evaluation class, and the leaf nodes that had to be overriden in order to react properly when connected to another gene fragment.

Much computational effort when running GFP is consumed by searching the space for fragments that meet certain location conditions. One method of improvement in this field is to divide space into smaller areas (cells), and keep information about the nodes that are currently located within each cell. Using such information only selected cells could be used for the search, which should speed up the algorithm. However, experiments with this feature implemented did not confirm such improvement. Spatial distribution analysis (see chapter 8.2.1) showed a possible reason for this observation. It was observed that in vast majority of cases, nodes are distributed evenly around the central point. This results in many cells being empty, while others are highly congested. If area covered by each cell is big, the number of fragments in each cell is still big as well, whereas if it is small, the overhead related to maintaining the information about fragments belonging to each cell beats the benefits of their use. In the end, since speed was not the most expected feature of GFP, the above mentioned mechanism was not used for experiments.

6.4 Final state

The GFP implementation is available as a modified ECJ framework. It can therefore be run just like any other problem implemented in ECJ, calling the main ECJ class and providing the appropriate user configuration file. Three such files are provided, thesis_parity.params, thesis_regression.params and thesis_ant.params, for each of the problems respectively.

7 Modularity and benchmarks

Before running the experimental part of research, problems needed to be selected on which GFP would be tested. Although there is a variety of benchmarks for GP, such as Santa Fe ant trail, lawn mower or symbolic regression, the choice was not trivial because GFP was designed with specific hypothesis on mind - that it supports emergence of modularity in problems that it solves. It would therefore be desired to select problems on which this feature could be observed best. Even more, it would be most convenient if the modular structure of the problem could be assessed *before* running the algorithm, so that certain expectations of algorithm behaviour could be formed and verified experimentally. To select problems with clear modular structure, the definition of modularity should be revised first.

In [9] a precise definition of modularity for genetic algorithms is given. In general, it defines a module as a set of variables that are worth optimizing independently of other variables in the problem. If a problem contains modules that themselves contain modules as well, it is called *hierarchical*. A classic example of hierarchical problem is the *Hierarchical If and only IF* (HIFF) problem. It has a recursive fitness function: If the bit string being considered consists of all zeros or all ones, the fitness of the string is equal to the length of the string, otherwise it has a fitness of 0. This same criteria is then applied recursively on each half of the string, until it can be subdivided no further. Adding the fitnesses of all substrings together yields the fitness of the whole [15]. The problem of using benchmarks such as HIFF is that they are GA problems, involving variables and their values and cannot be therefore applied directly to GP methods such as GFP. The modularity definition mentioned above, however, could be translated into similar definition for GP - modules would be blocks of code that could be optimized independently from the rest of the program.

As far as GP modularity is concerned, [18] describes 3 approaches. *Function definition* was introduced by Koza in [14]. It is an attempt to separate blocks of code in order to let them evolve independently; each such block is called a *function* and can be parametrized by one or more arguments. Functions are furthermore used in an approach called *Automatic Function Definition*, in which declarations of functions (that is, the number of arguments it accepts) are defined before the experiment, and what evolves is their definitions as well as the main program that utilizes them to calculate the overall program result. Second approach, called *module acquisition* [7], defines modules as parts of programs that work well as a whole and thus should not be modified. Module acquisition uses 2 special operators, *compression* for freezing a block of code and *decompression* for allowing it to be modified again. Third approach is called *adaptive representation* (AR) [19]. AR attempts to find commonly used blocks of code and extract them, so that they can be shared by other solutions. Such process can also be applied to combinations of fragments, in a way that allows to build more and more complex subroutines.

With regard to the three approaches described above, how do they identify modules in a solution? Module acquisition applies the compression and decompression operators to random pieces of code, assuming that if the fragment is in fact a separate module, it will prevail in the population, and it will slowly be replaced otherwise. Adaptive representation uses heuristics or statistical information to estimate which fragments of code should be extracted, in both cases basing it mainly on fitness distribution. Adaptive representation through learning (ARL) [18] uses 2 such heuristics, called differential fitness and block activation. Both in module acquisition and adaptive representation, a module is understood as a block of code that in course of evolution is detected to be useful as a whole and as such should be protected against unnecessary modifications. ADF takes a direct approach by defining the function declarations beforehand, and allowing the crossover operator to exchange parts of code only belonging to similar functions. A module is therefore a block of code that may evolve but is limited by its declaration.

In GFP the notion of modularity is most similar to module acquisition and adaptive learning, since the introduction of the gene fragments is expected to make useful blocks of code less prone to destructive modifications. It is also, in some way, similar to the GA definition as gene fragments evolve independently from each other. Module acquisition used in cartesian genetic programming (CGP) has been tested on the even parity problem, mainly because it is relatively easy to visualize any block of code [13]. ARL has been analyzed using a version of the Pac-Man problem, modified to make visualization and code analysis easier [18].

The Pac-Man problem is unfortunately unavailable in the ECJ environment. Since there is no GP problem with a clear modular structure, the set of benchmarks used for analyzing the GFP has been defined as follows:

- the symbolic regression problem, for it is easy to implement and analyze
- the ant trail problem, for it is expected that it might promote emergence of modules defining behavour in different situations, for example in cases where there is or there is no food ahead
- the even parity problem, for it is easy to visualize

Not always all experiments are done using all these benchmarks, it depends on the goal of the experiment in each case.

8 Experiments

8.1 GFP parameters analysis

First part of investigation is devoted to the analysis of algorithm parameters. Most steps of the algorithm may be conducted in one of a few manners, as it was described in chapter 4. These options will be referred to as *dimensions*, since in most cases they are independent from each other, meaning that any combination of them represents a valid configuration of GFP. Overall, there are 4 dimensions of GFP:

1. Fragment selection method: n-closest / roulette / all-in-range / random
2. Solution expansion method: roulette / tournament
3. Award method: classic / inertial / asymmetrical / decay
4. Build-stop method: size

Additionally, some dimensions need to be further parametrized, for example "n-closest" fragment selection method needs to have defined the number of closest nodes ("n"). The goal of this experiment is to run GFP with different combinations of dimension values and assess its results. Two characteristics were measured: solution quality and algorithm speed. Solution quality is the adjusted fitness of best solutions generated in each of run of the algorithm. In ECJ, that is a real value from 0.0 (exclusive, worst) to 1.0 (inclusive, best), for any problem domain. Algorithm speed was evaluated by measuring time of each run of the algorithm.

Algorithm was tested on two benchmarks: symbolic regression (quartic) and Santa Fe ant trail. Table 1 presents the most significant GP settings that were used. In particular, crossover was turned off and mutation was used instead. This was done to limit the influence of genetic operators on the algorithm, in order to assess its effects with bigger confidence. Maximum depth of individual was also limited, in order to promote linking solutions from more than one fragment. In classic GP the limit was 17 (after crossover or mutation, 6 after generating new ones) which in both problems would allow to develop ideal solutions consisting of only one fragment. As for the mutation maximum depth limit, three values were tested that allowed small (maximum depth 3), average (maximum depth 5) or large size fragments (maximum depth 7).

As far as spatial information is concerned, experiments were run with fragments located in two-dimensional space, that is each location was a vector of a length of 2.

Overall, the following configuration values were tested:

- *fragment selection method: 1-closest, 3-closest, 5-closest, random.* The n-closest selection method was tested in 3 variants, in which each leaf was adding 1, 3 or 5 closest fragments to the candidate set. Random method was also tested to allow comparing the influence of fragment selection method on overall solution fitness. After initial runs the all-in-range method was not included, since it depended too much on the spatial distribution of fragments. In many cases fragments were grouped closely on small areas of hyper-space, which often made the algorithm terminate solution construction phase very early due to empty candidate set. More information on that can be found in chapter 8.2.

No	*Setting*	*Setting description*	*Value*	*Remarks*
1	*pop.subpop.0.species.pipe.source.0*	*genetic operator applied to individuals*	*MutationPipeline*	*by default CrossoverPipeline is used instead*
2	*pop.subpop.0.species.pipe.source.1*	*genetic operator applied to individuals*	*ReproductionPipeline*	
3	*pop.subpop.0.species.pipe.source.0.prob*	*probability of selecting first operator*	*0.9*	
4	*pop.subpop.0.species.pipe.source.1.prob*	*probability of selecting second operator*	*0.1*	
5	*breed.reproduce.source.0*	*selection method used by reproduction*	*TournamentSelection*	
6	*gp.koza.mutate.maxdepth*	*maximum tree depth after mutation*	***3 / 5 / 7***	*default is **17**, this is the one of the parameters that limit the size (depth) of individuals - solution fragments in case of GFP*
7	*gp.koza.half.max-depth*	*maximum tree depth when generating new tree*	***3***	*default is **6**, this is also to limit the size of solution fragments*
8	*generations*	*maximum number of generations in each run*	*40*	
9	*pop.subpop.0.size*	*number of individuals in population*	*512*	*default is 1024*
10	*eval.problem.size*	*number of points generated for symbolic regression problem*	*20*	*applies to symbolic regression only*

Table 1: Selected GP settings used in GFP parameter analysis

		Classic		Decay 0,01		Decay 0,005		Inertial	
Selection	Fragment size	Quality	Time [s]	Quality	Time [s]	Quality	Time [s]	Quality	Time [s]
roulette	3	**0.52**	40.9	0.42	**33.9**	0.41	42.0	0.44	35.5
roulette	5	**0.48**	154.8	0.28	154.0	0.34	154.8	0.15	**145.5**
roulette	7	**0.46**	273.5	0.29	269.7	0.33	269.7	0.12	**259.7**
tournament	3	**0.52**	42.9	0.41	41.5	0.41	**34.2**	0.40	38.4
tournament	5	**0.49**	166.2	0.27	166.9	0.37	**162.9**	0.15	189.1
tournament	7	**0.50**	278.9	0.25	272.4	0.37	**259.4**	0.14	300.6

Table 2: Results for: regression problem, 3-closest fragment selection

- *solution expansion method: roulette, tournament.* Both options were tested. Roulette method was selecting fragments based on their distance, from the whole candidate set. The size of the tournament could itself be a parameter as well but for simplicity was assumed to be 7, which is the default value for tournament selection in Koza GP.
- *award method: classic, inertial, decay 1%, decay 0,5%.* The classic award method does not have any parameters. The inertial method had $p = 0,7$, which means it took the new fitness with weight 0,7 and old fitness with weight 0,3. Other values of p had also been initially tested ($p = 0,5$, $p = 0,9$) but did not influence the results significantly. The decay award method was taken with 2 values of the decay rate: 0,01 and 0,005. Asymmetrical method was not included in this experiment, since initial results showed it was inferior to a similar "inertial" method.
- *build-stop method: size.* When discussing the possible methods of detecting when to stop building the solutions, methods other than by size required well tuned parameters that would provide best results for a given configuration. In practice, it turned out to be very difficult to properly adjust such values, since the results calculated by GFP already depend on a number of other parameters. In the end, only the size method was used, for it was most robust, at the acceptable cost of possibly redundant computational effort.
- fragment size: 3, 5, 7. The maximum depth for a gene fragment.

Each combination of configuration values was run 20 times for each problem, median values were calculated to avoid negative influence of outliers. With a similar configuration, the classic GP obtains the following results:

- symbolic regression problem: quality 0.79, time 7.21 s
- ant trail problem: quality 0.052, time 4.10 s

Results in tables 2 and 3 show clearly that classic fitness award method provides best average fitness quality for any combination of other parameters. Award method with penalization of neutral fragments (decay) produces significantly worse results, the worse the stronger the penalization rate, similarly to inertial method that uses a simple recursive filter to include previous fitness values when assigning new ones. These differences are more visible for the regression problem, where the classic method gives results even 3 times bigger than inertial (0.50 vs 0.14 for tournament with biggest fragments).

		Classic		Decay 0,01		Decay 0,005		Inertial	
Selection	Fragment size	Quality	Time [s]	Quality	Time [s]	Quality	Time [s]	Quality	Time [s]
roulette	3	**0.026**	**147.2**	0.018	149.1	0.024	147.9	0.023	154.8
roulette	5	**0.022**	**272.3**	0.018	314.5	0.020	322.1	0.019	279.2
roulette	7	**0.027**	511.0	0.016	491.1	0.025	**472.3**	0.020	565.1
tournament	3	**0.035**	153.8	0.017	145.6	0.022	**145.1**	0.021	155.0
tournament	5	0.020	306.5	0.016	**263.5**	0.019	270.4	**0.024**	281.9
tournament	7	**0.022**	538.8	0.017	**474.7**	0.020	509.8	0.020	535.5

Table 3: Results for: ant trail problem, 3-closest fragment selection

		1-closest		3-closest		5-closest		Random	
Selection	Fragment size	Quality	Time [s]	Quality	Time [s]	Quality	Time [s]	Quality	Time [s]
roulette	3	0.49	30.2	**0.52**	40.9	**0.52**	41.4	0.51	**11.1**
roulette	5	0.45	134.0	**0.48**	154.8	**0.48**	175.0	0.37	**22.8**
roulette	7	**0.51**	224.4	0.46	263.9	0.48	302.7	0.34	**32.3**
tournament	3	0.51	27.3	0.52	32.9	**0.53**	38.4	0.52	**11.1**
tournament	5	0.48	132.4	**0.49**	166.5	0.48	189.2	0.39	**24.3**
tournament	7	0.47	228.1	**0.50**	267.8	**0.50**	300.6	0.35	**30.1**

Table 4: Results for: regression problem, classic award method

		1-closest		3-closest		5-closest		Random	
Selection	Fragment size	Quality	Time [s]	Quality	Time [s]	Quality	Time [s]	Quality	Time [s]
roulette	3	**0.043**	124.4	0.026	147.2	0.021	173.0	0.020	**11.8**
roulette	5	0.022	224.7	0.022	272.3	**0.026**	329.2	0.022	**14.5**
roulette	7	**0.040**	438.8	0.027	511.0	0.026	607.7	0.020	**17.5**
tournament	3	0.030	123.9	0.035	153.8	**0.038**	174.5	0.030	**11.6**
tournament	5	0.025	231.1	0.020	306.5	**0.043**	295.5	0.021	**13.3**
tournament	7	0.023	441.3	0.022	523.8	0.022	598.2	**0.024**	**15.4**

Table 5: Results for: ant trail problem, classic award method

When looking at the fragment sizes in the regression problem, in table 4, it can be noticed that for the smallest fragments, of the maximum depth of 3, the fitness achieved remains similarly good even if purely random method is used. A probable explanation is that for smaller fragments their diversity is smaller compared to other cases and the whole algorithm becomes similar in behaviour to classic GP. For this reason, bigger fragment sizes limits should be used in further analysis - a value of 5 was selected taking into consideration the time consumption increasing significantly with the size of fragments.

As far as the size of neighbourhood is concerned (in terms of numbers of fragments for the N-closest method), all values give similar results for the regression problem (table 4). The value of 3 was selected for later analysis, since it was slightly better than for N=1, and its execution took less time than for N=5. Surprisingly, the results for the ant problem, table 5, were not very consistent. The solutions fitnesses achieved varied from 0.21 to 0.43, with the random fragment selection method providing very similar results. This is most probably to the fitness landscape characteristics for the ant trail problem, in which there is a certain set of solutions which are evaluated much higher than the rest, as a result of what median fitness may change rapidly depending on the number of such solutions appear in the examined set. A possible remedy for such situation is to repeat this experiment with a much bigger number of runs.

For the regression problem, GFP achieves solution fitness of 0.52, compared to 0.79 achieved by GP. For the ant trail problem, the results are 0.043 (GFP) vs. 0.052 (GP), if both algorithms are run for 40 generations and 512 individuals in a population. Though GFP in both cases is not as good as classic GP, the difference is smaller for the ant trail problem. This confirms the initial expectations that the ant trail problem may be more suited for GFP than the symbolic regression problem. It has to be noted however that default settings for both problems in ECJ include 1024 individuals in population - when run with such configuration, GP achieves fitness of 0.5, compared to only 0.045 achieved by GFP, which additionally is many times slower than classic GP. For the even-3 parity problem, classic GP always solved it (fitness 1.0), while GFP would always achieve fitness of 0.5. For the even-4 parity version, GFP achieves fitness no higher than 0.2.

8.2 Modularity analysis

Second part of investigation is dedicated to analyzing the course of algorithm and determining if the hypotheses about automatic problem decomposition is true. In other words, the goal is to determine to what extent does GFP detect modules of problems.

Measuring modularity is not an easy issue mainly because there is no precise definition of modularity in GP solutions (see chapter 7). The first approach is to investigate the distribution of locations of best solutions in a run. Solution are assembled from gene fragments which, hopefully, may represent modules of the solutions. As fragments are selected based on their locations (locations of their root nodes, precisely), we expect evolution to form groups of gene fragments with regard to their locations, where fragments of each group serve similar function in solutions and have similar chances of being selected during assembly. Analyzing the spatial distribution of fragments could help to decide if such gene fragment groups form themselves, and to what extent if they do.

8.2.1 Gene fragments spatial distribution

Analysis was performed on all three benchmarks. Maximum number of generations was 40 and population size was 512. Maximum fragment depth was 5. Single runs were analysed because adding fragment locations from separate runs would rather lead to loss of information - since the space is symmetrical, any particular groups of fragments that form within each run would not be visible when adding data from many runs. Each of the runs selected for analysis had been compared before to other runs to assure that its characteristics are more or less representative to the whole.

First diagrams present locations of fragments, separately for generations 1-20 and 21-40. This should give a notion of overall gene fragment spatial distribution. Because many fragments are probably not used in constructing solutions, similar diagrams have been done that present only locations of 50% and 5% of fragments that were most used in given generation. Comparation of these diagrams should show which regions contain frequently used fragments, which may be a result of grouping by functionality among the fragments. In these diagrams, the X and Y axis represent 2-dimensional space, while Z axis shows how many fragments are located in each space unit. The center of the space is located in point (100, 100) and locations are scaled so that the average distance to the center of the space is 50.

Figures 8, 14 and 21 show the usage frequency analysis. For each generation, it is counted how many times all fragments are used in total, then the same for the 50% most frequently used fragments, and for the 5% most frequently used. For each problem, the number of fragments used for the best solution in each generation is also calculated. Finally, the average and best fitness for each generation is presented, to assess the overall quality of generated solutions.

For the regression problem, when analysing the locations of fragments within the first 20 generations (figure 4), it can be noticed that fragments are dispersed rather equally around the center point, with most fragments located at the edge of the "circle", that is around 50 units from the center. Some areas contain more fragments than others, those are marked red on the figure. On figure 5 only 50% of fragments that were most frequently used are marked. When zooming and rotating the figure, it can be observed that mostly fragments located in the central area are missing compared to the previous figure. That tendency is confirmed on figure 6, which shows only 5% most used fragments within the first 20 generations, and on figure 7 which shows 5% most popular fragments within generations 21-39.

Similar observations can be made for the ant trail problem - almost all fragments are located within the distance of 50 units from the center point (figure 11), though groups of fragments are more clear than in the regression problem. For the parity problem (figures 17-20) fragments are located not only on the circle 50 units from the center point but also a bit closer and further. 3 groups of fragments are clearly visible, both for the whole population and for the most frequently used individuals.

The fragments "popularity" distribution graphs (figures 8, 14 and 21) show that the distribution stays constant throughout the run for each problem. All fragments are used altogether around 2500 times for each generation, most of which is due to no more than 50% of individuals. This shows that the neutrality level in GFP is quite high, as close to half of individuals are used very rarely or not used at all. On the other hand, the 5% most used fragments are used in no more than 10-15% of cases, which suggests that there is quite a variety in the fragments selected for solution construction.

As far as number of fragments used to construct best solutions, this a problem-dependent characteristic. For regression problem it can be almost any number between the minimum 1 and maximum 7 fragments, whereas in parity and ant problems highly evaluated solutions consist typically of no less than 4 fragments. Within these ranges, the numbers vary. This can be due to the specific procedure of determining the size of solutions (the build-stop method, see chapter 4.3.3), in which solutions are always constructed of maximum number of fragments, in this case 7, but they are evaluated after each new fragment is added, as if 7 solutions of various sizes were constructed. If during the evolution a fragment is added to the solution that improves its fitness, the number of fragments comprising the best solution may increase. If, however, a well working fragment gets moved away due to unsuccessful attempts to create solutions with a different set of fragments, it is no longer taken into consideration and the number of fragments creating the best solution decreases.

Such situation seems to be confirmed by the fitness graphs (figures 10, 16 and 23). The average fitness is similar throughout the evolution but best fitness changes rapidly, both rising and falling. This may suggest that even when good solutions are constructed, fragments take part in other attempts as a result of which they are relocated, and the good combination is broken.

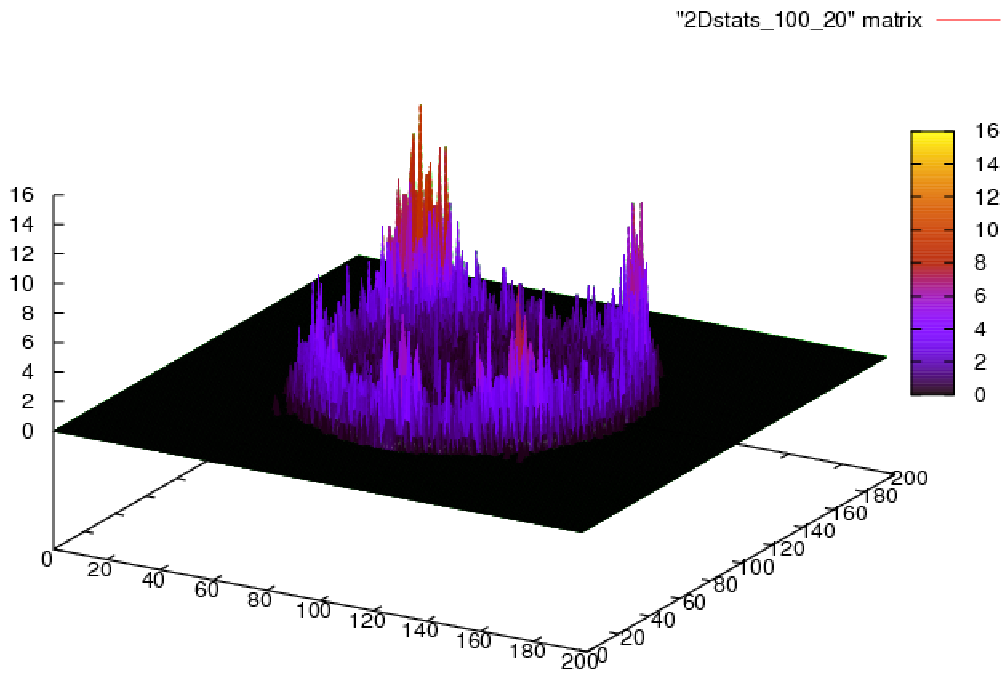


Figure 4: Regression problem, locations of all fragments within generations 1-20

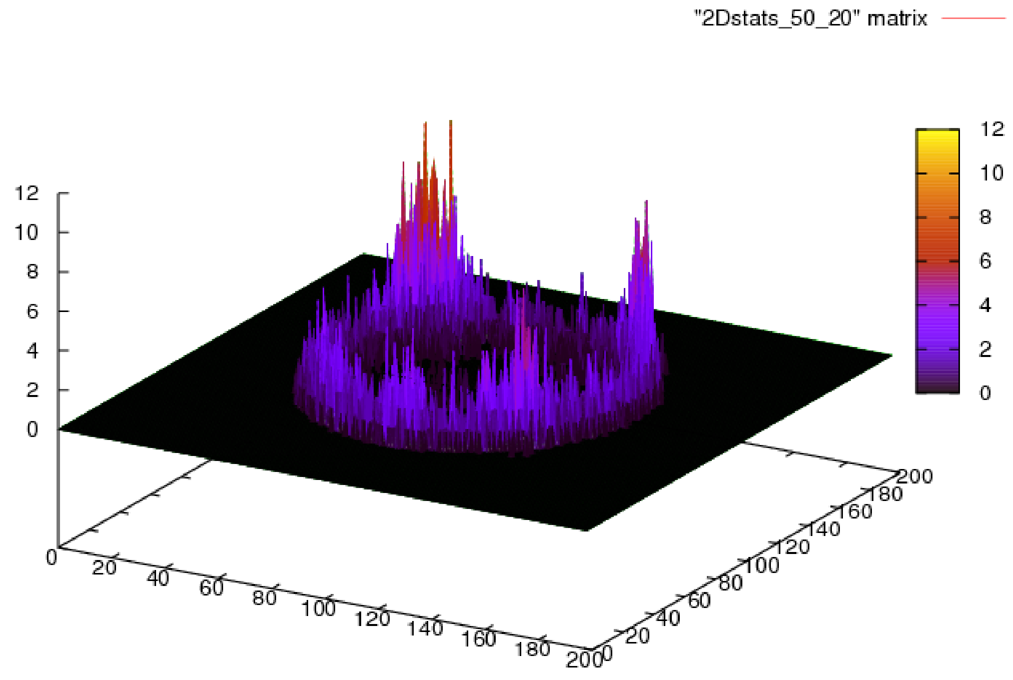


Figure 5: Regression problem, locations of 50% most frequently used fragments within generations 1-20

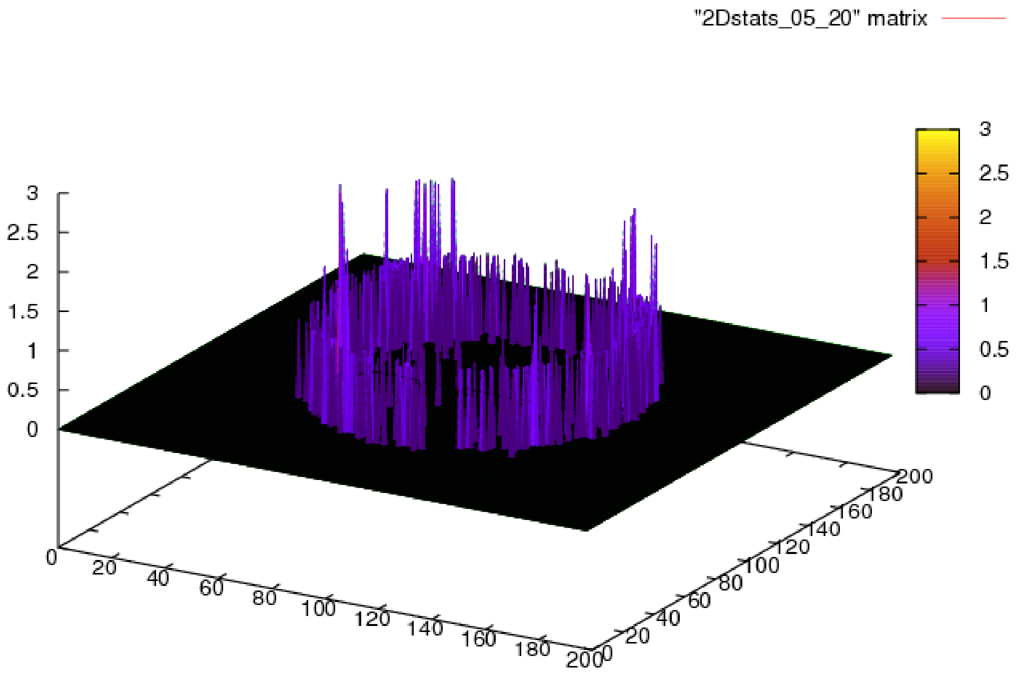


Figure 6: Regression problem, locations of 5% most frequently used fragments within generations 1-20

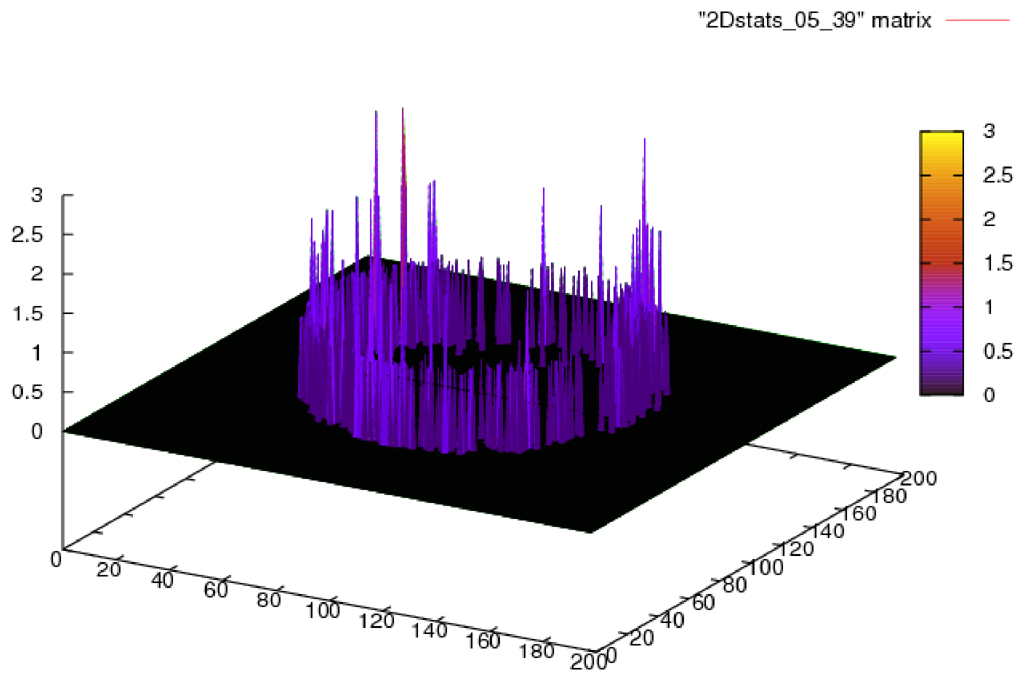


Figure 7: Regression problem, locations of the 5% most frequently used fragments within generations 21-39

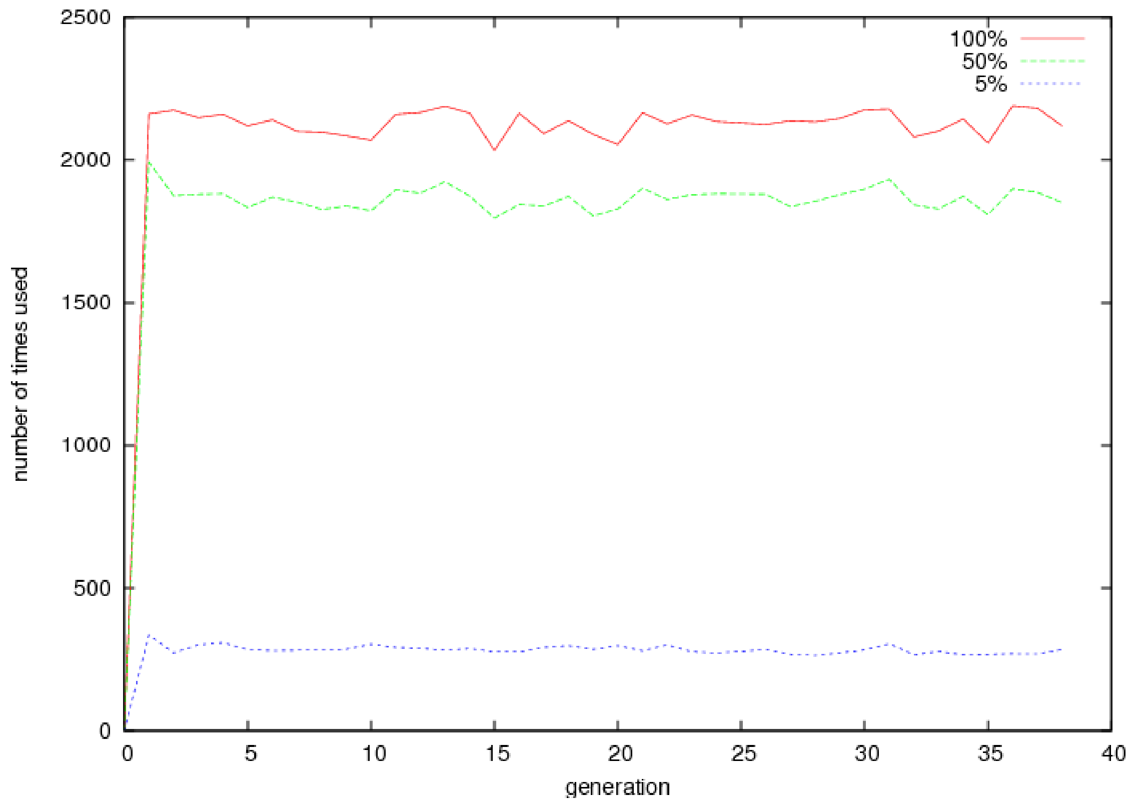


Figure 8: Fragment popularity distribution for regression problem

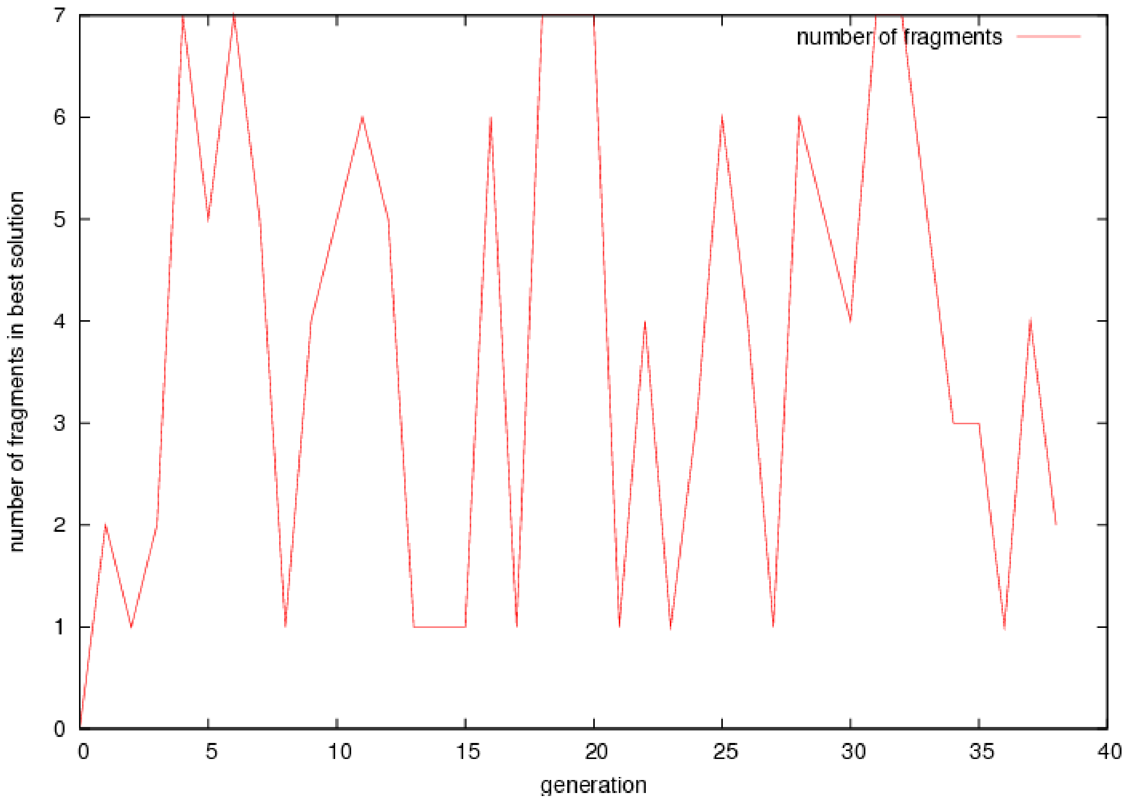


Figure 9: Number of fragments in best solutions of each generation, regression problem

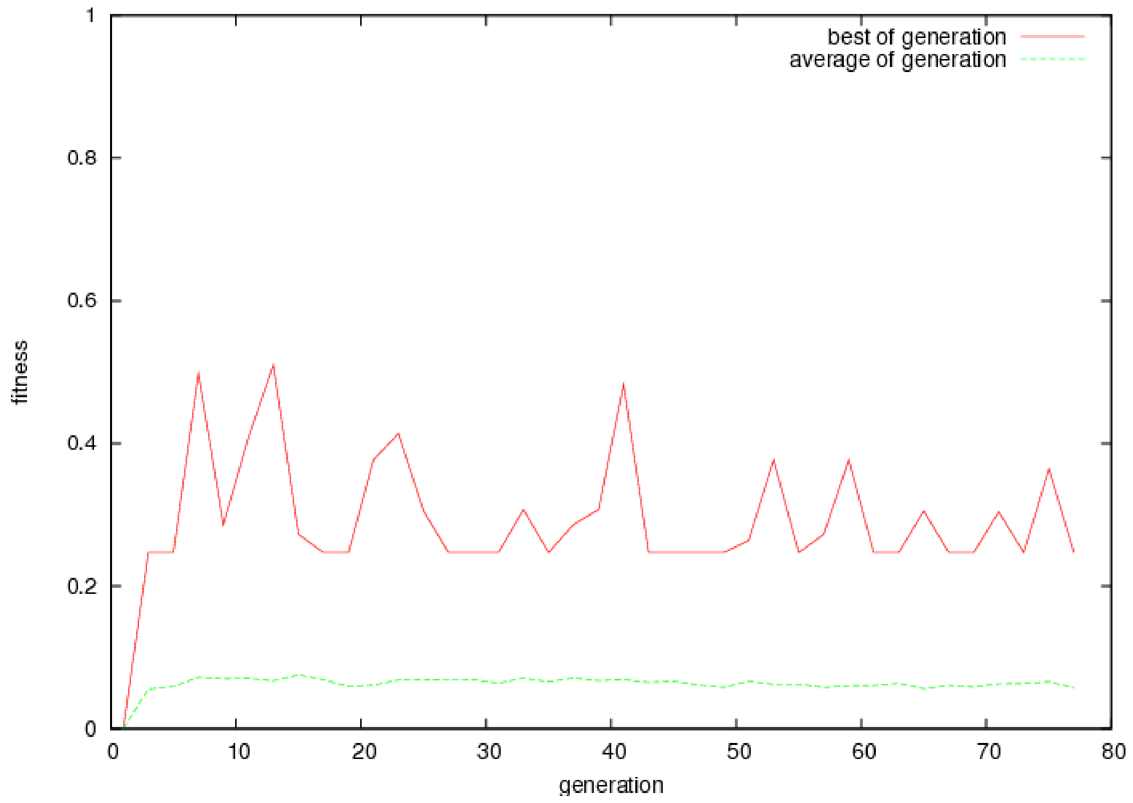


Figure 10: Average and best solution quality achieved by GFP for regression problem

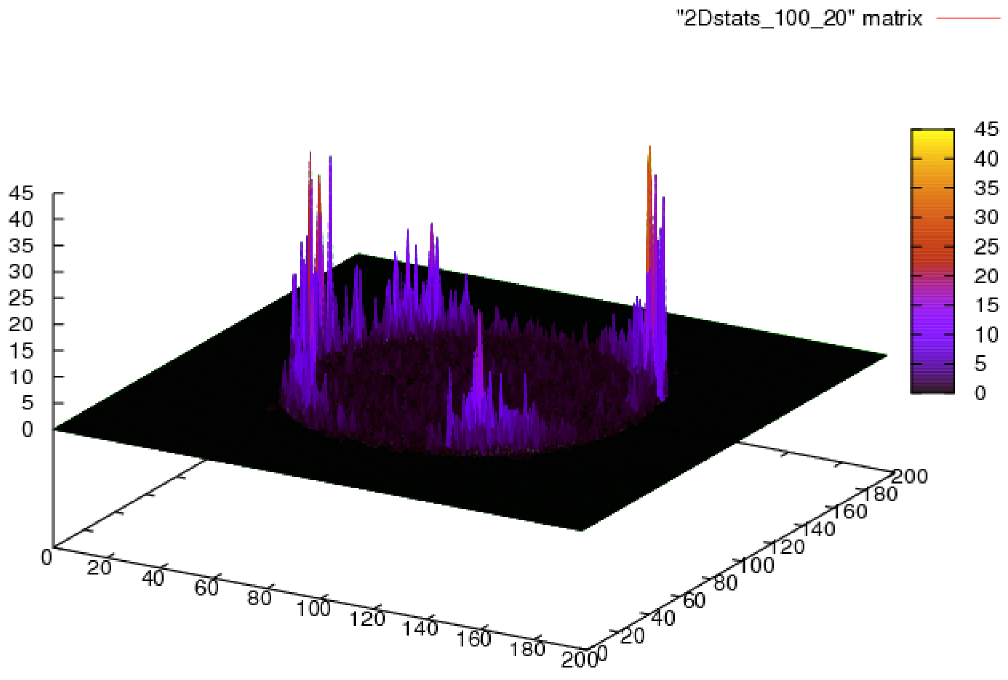


Figure 11: Ant trail problem, locations of all fragments within generations 1-20

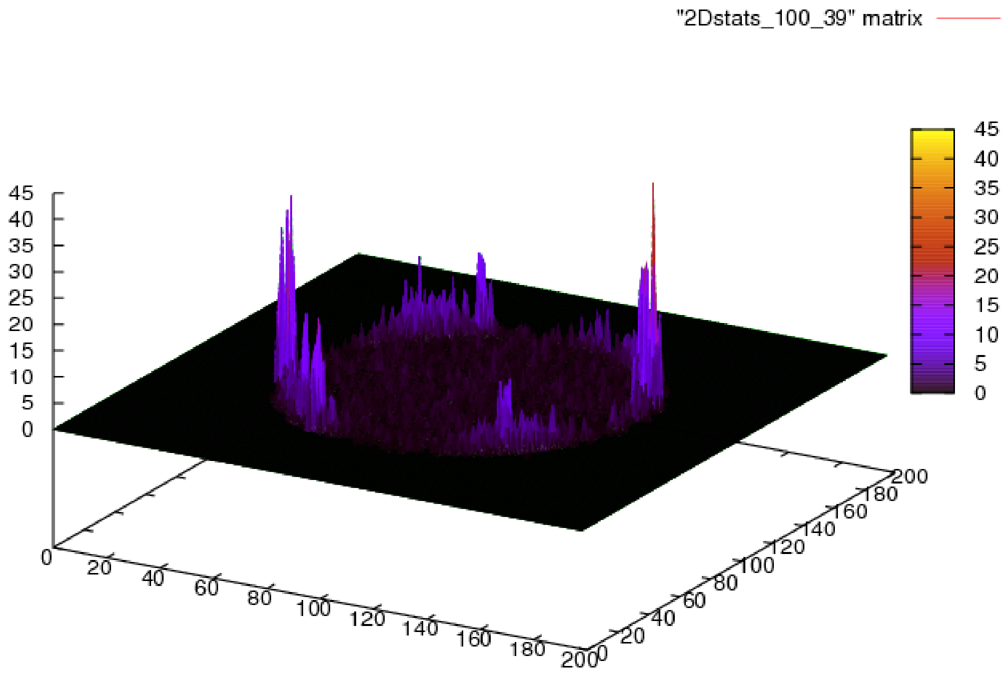


Figure 12: Ant trail problem, locations of all fragments within generations 21-39

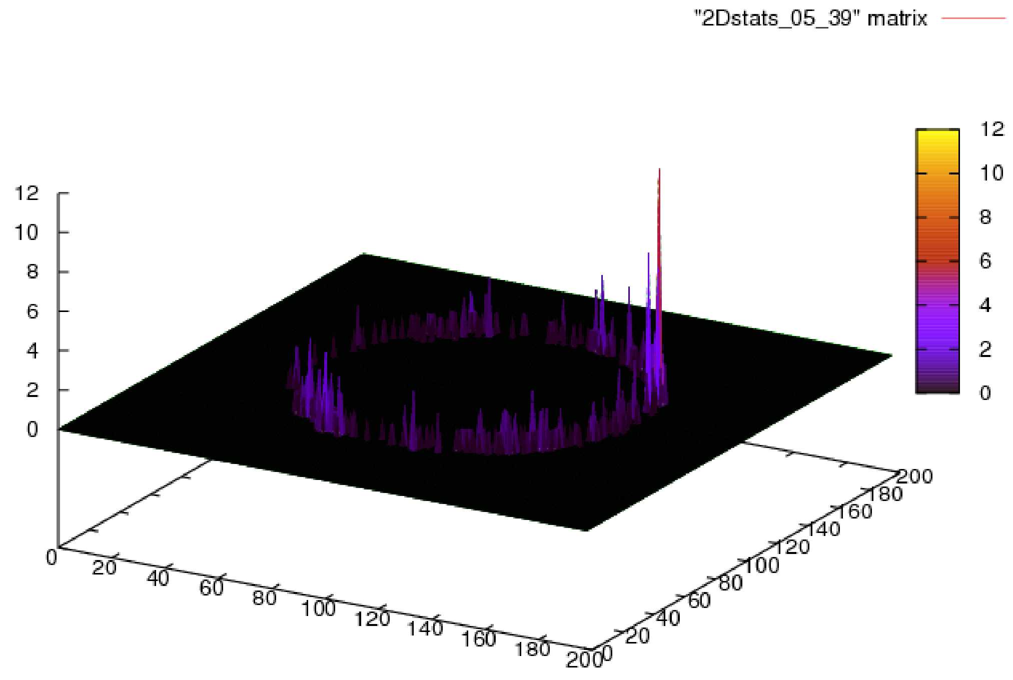


Figure 13: Ant trail problem, locations of the 5% most frequently used fragments within generations 21-39

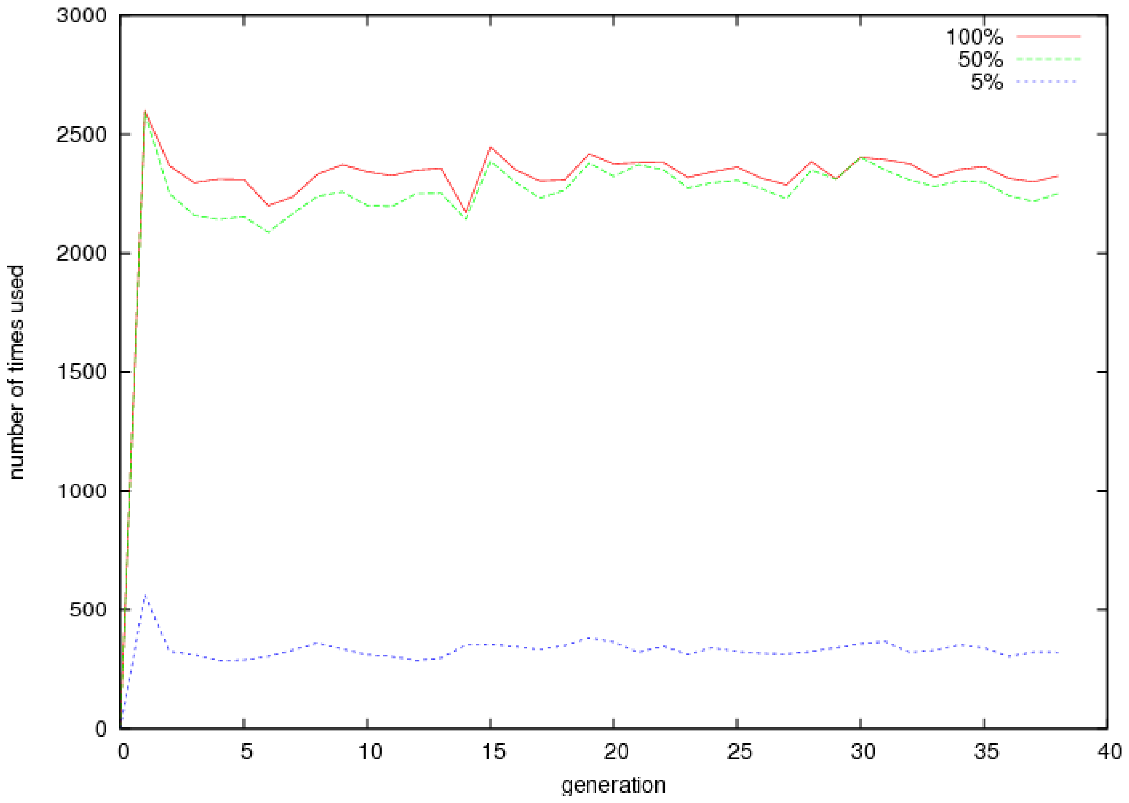


Figure 14: Fragment popularity distribution for ant trail problem

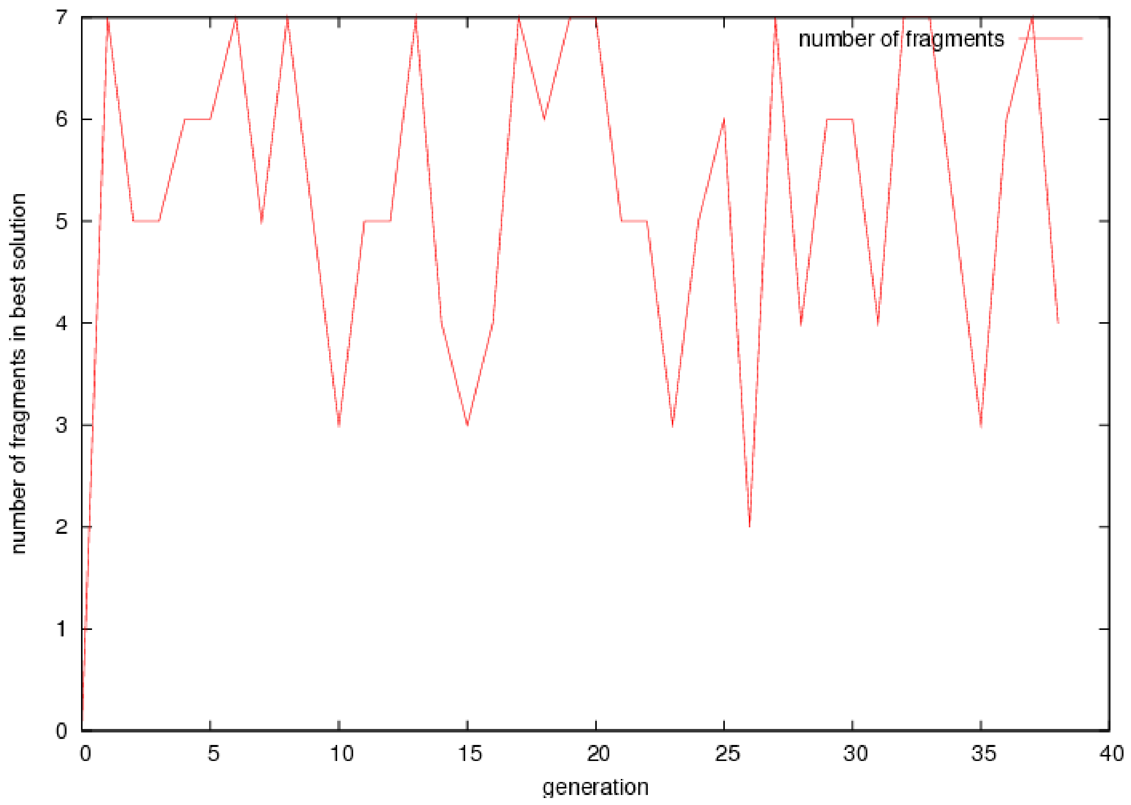


Figure 15: Number of fragments in best solutions of each generation, ant trail problem

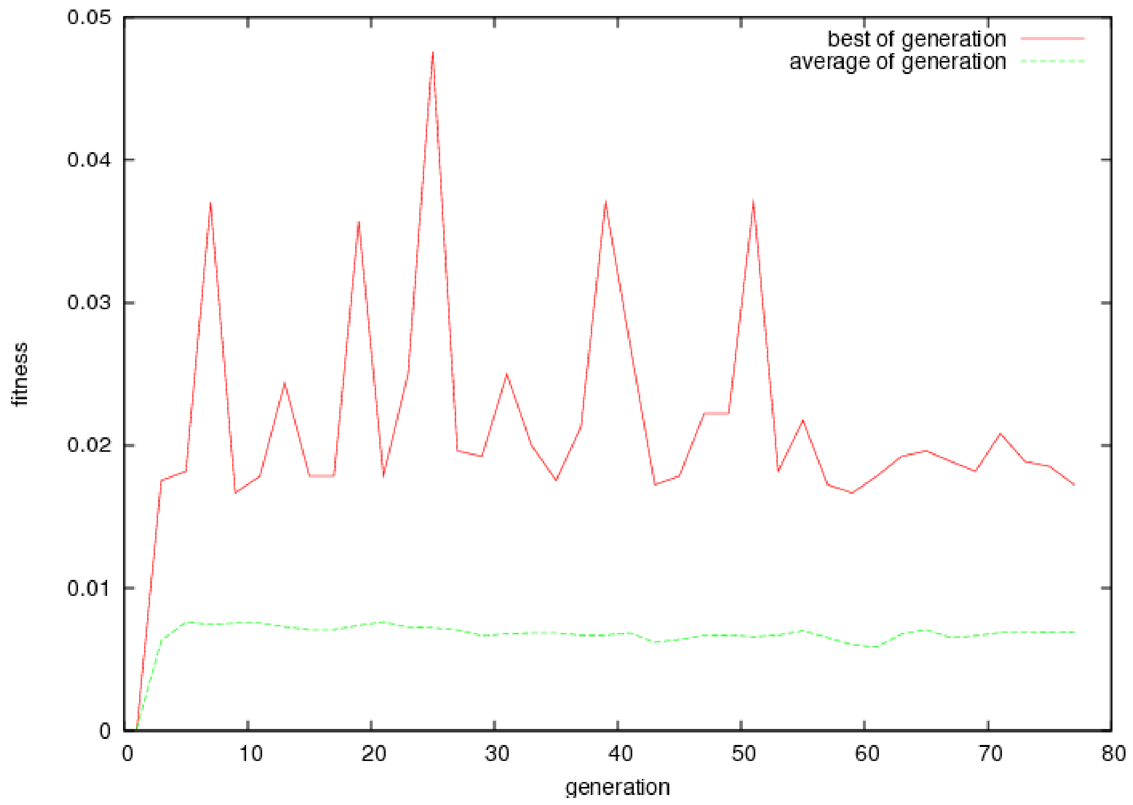


Figure 16: Average and best solution quality achieved by GFP for ant trail problem

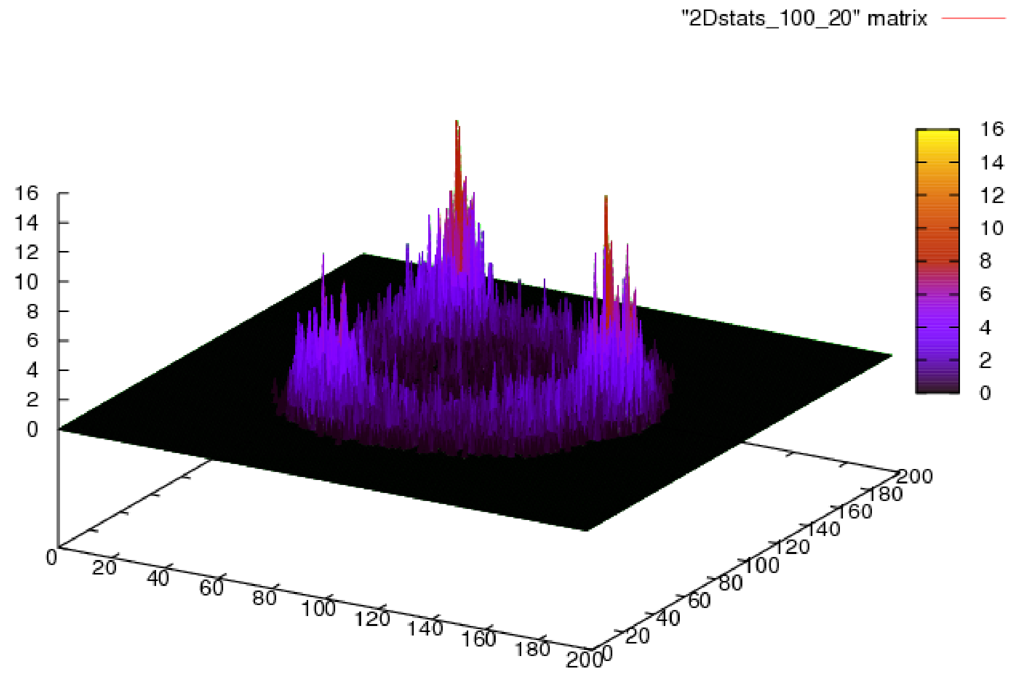


Figure 17: Parity problem, locations of all fragments within generations 1-20

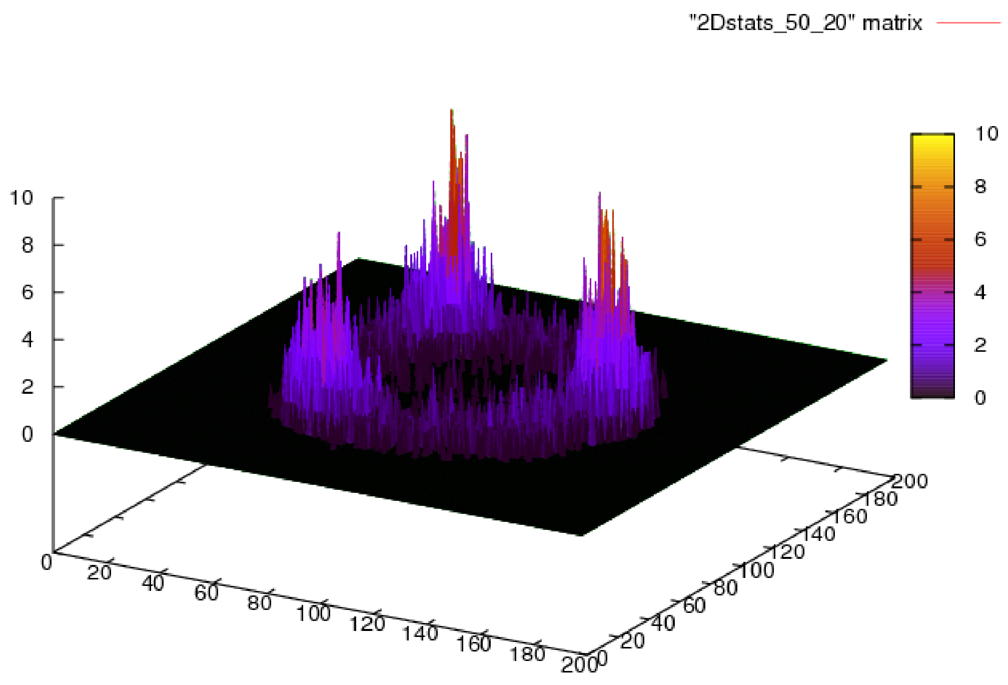


Figure 18: Parity problem, locations of 50% most frequently used fragments within generations 1-20

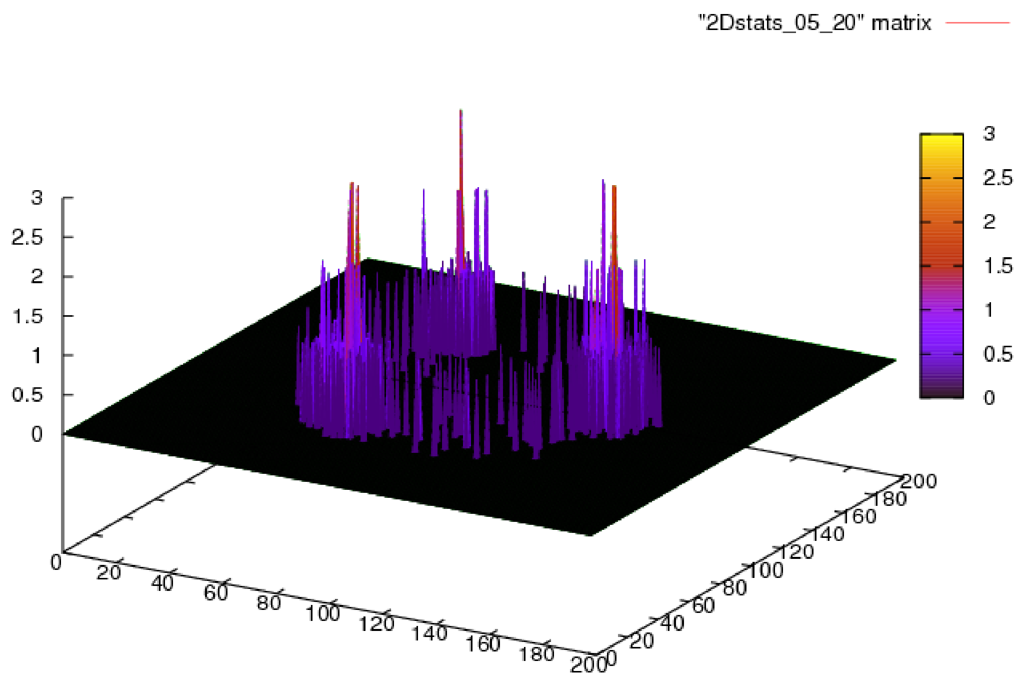


Figure 19: Parity problem, locations of 5% most frequently used fragments within generations 1-20

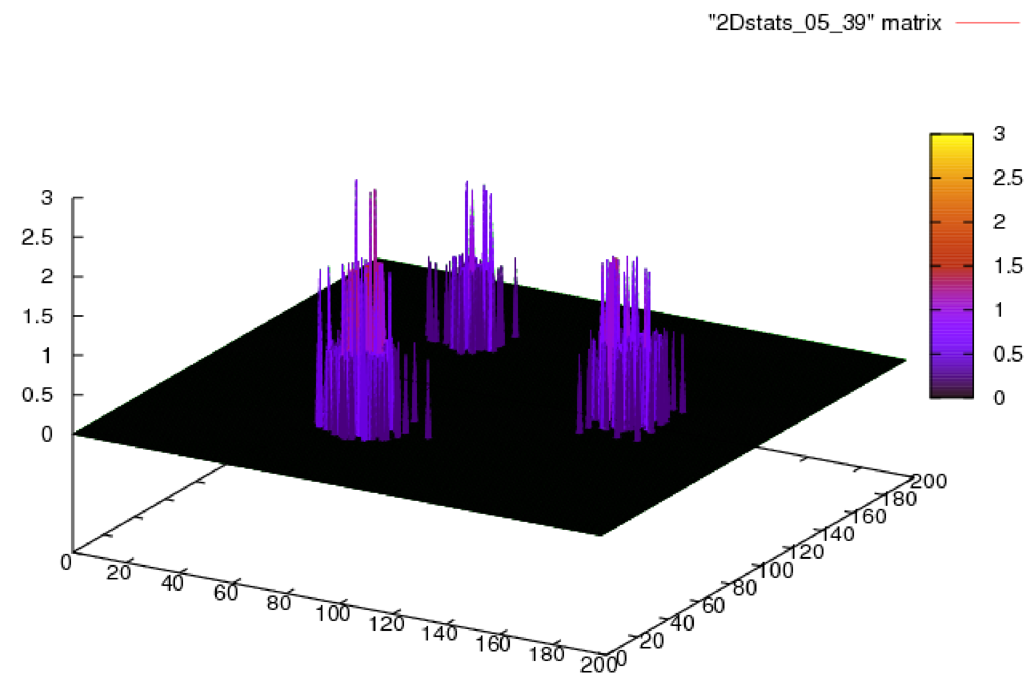


Figure 20: Parity problem, locations of the 5% most frequently used fragments within generations 21-39

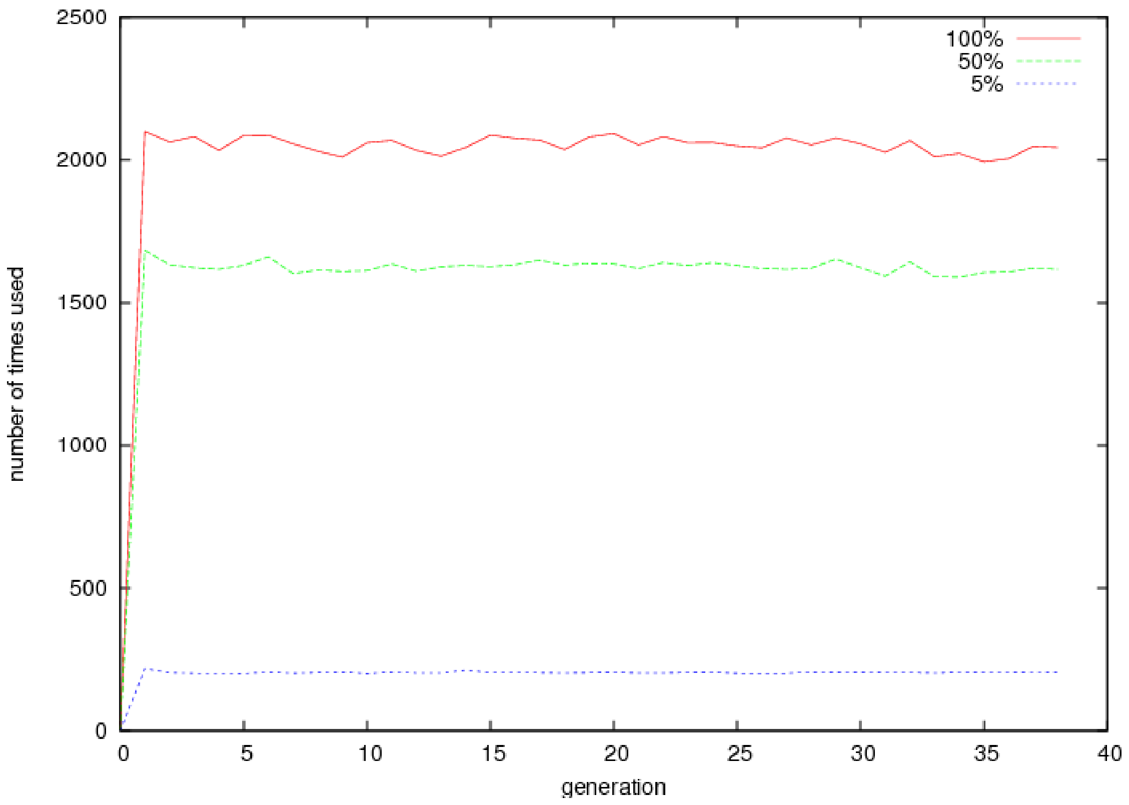


Figure 21: Fragment popularity distribution for parity problem

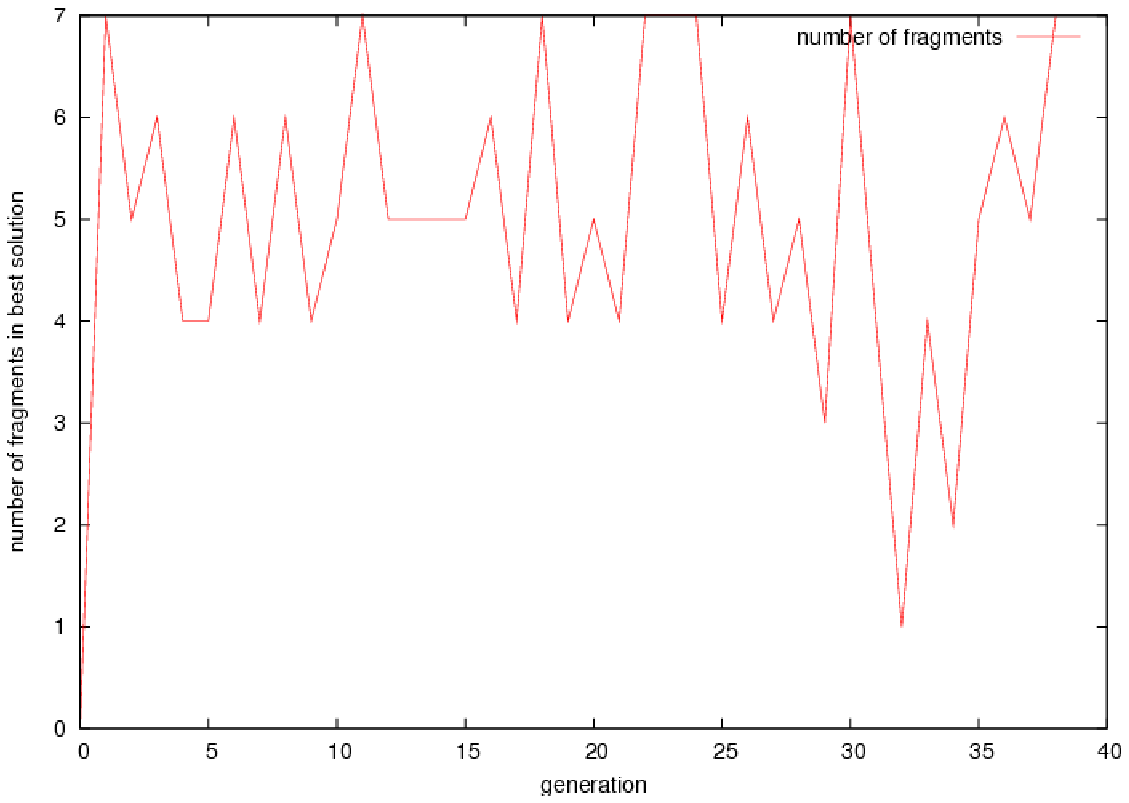


Figure 22: Number of fragments in best solutions of each generation, parity problem

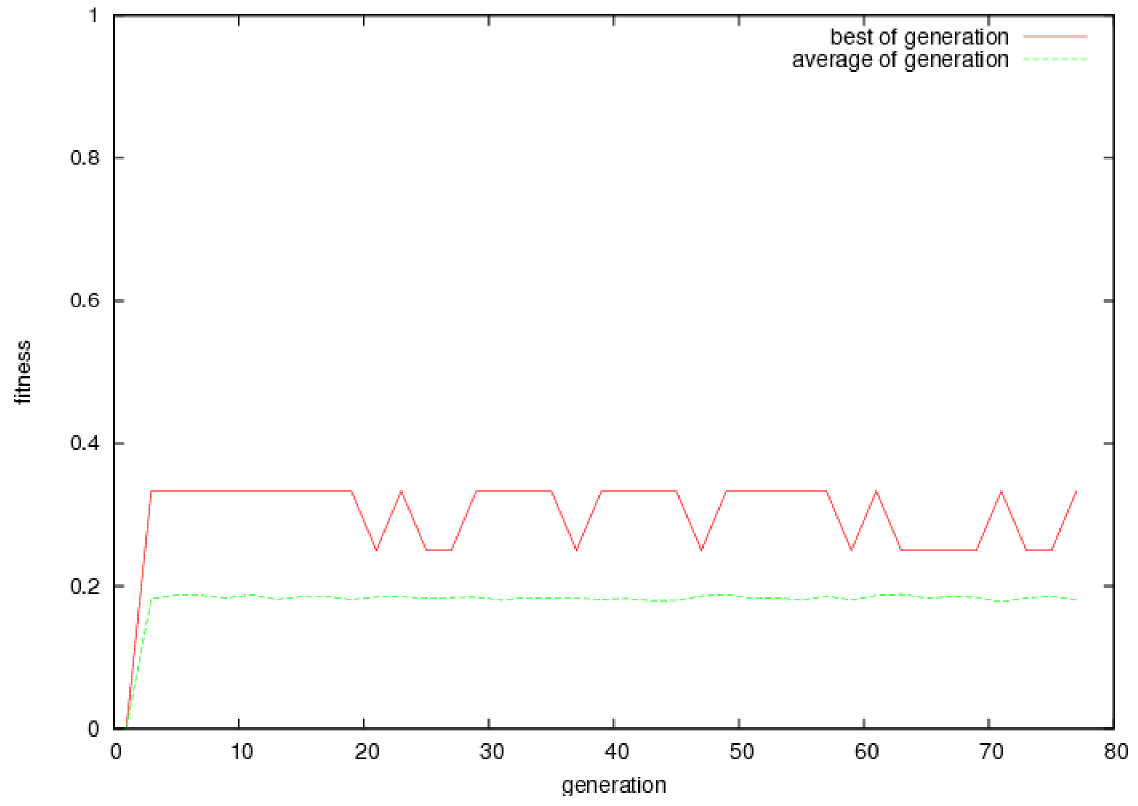


Figure 23: Average and best solution quality achieved by GFP for even-3 parity problem

A very interesting observation from the fragment distribution analysis is that for all examined problems the fragments are grouped at the edges of the area, in maximum distance from the center point that is allowed by the space scaling method (which keeps the average distance constant). Such situation had not been anticipated and it may have negative influence on the quality of algorithm, for two reasons: firstly, it decreases the two-dimensional space to almost one-dimensional - that is because for each fragment searching for closest fragments can be limited to searching “left” and “right” on the circle around the central point. Secondly, just selecting the first fragment of the solution may be practically random if many nodes, located on opposite sides of space, have very similar distance to the central point. Apart from that, such uneven distribution was the reason why searching for neighbour nodes within a given proximity failed (the “all-in-range” approach, see chapter 4.3.1).

When investigating the reasons for such situation, it has to be noted that after every solution construction fragments locations are modified, so that the children fragments are moved towards parent fragments in a degree corresponding to solution fitness. That means, that *evaluating solutions is a force that brings fragments closer to the space centre*, which is countered by scaling the whole population so as to keep the average distance to the center constant. The combination of these factors does not need to cause the fragments to be located only at a circle around the center point - one can imagine a situation when good fragments are located closer to center while worse or unused are located further.

If the fragments close to the center are not moved away from the center, that suggests that they are pushed

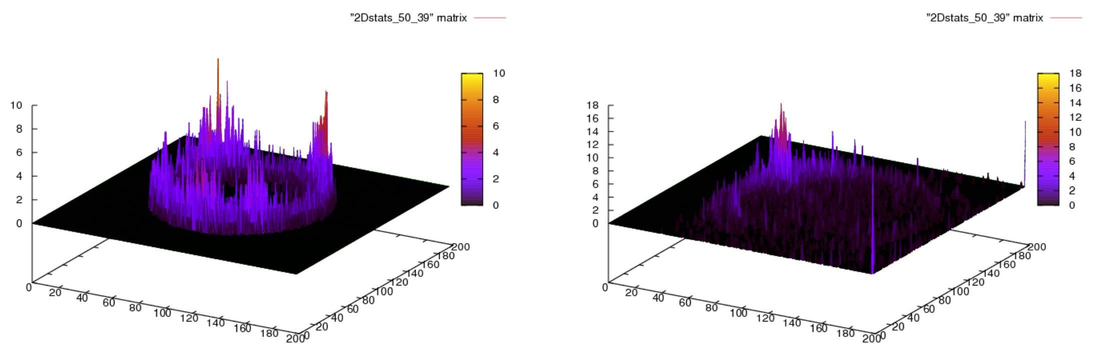


Figure 24: Classic and promotoclose methods comparison, 50% most frequently used fragments in regression problem

out of the population because of their low fitness. Such situation could be caused by imbalance in function that assigns fitness to fragments - if participation in bad solutions brings fitness down much more than participation in good ones raises it, fragments that are selected too often will have the worst fitness. And fragments located closer to the center have higher chances of being selected for solutions at least at the initial stage of experiments - if fragments are distributed randomly, statistically there is less fragments at distance 1.0 from center than, say, 3.0. To verify that hypothesis, a special fitness award method was introduced, called promoteclose - it is a modification of a classic award method, which calculates the fitness according to the equation:

$$f_p = f \cdot \frac{1}{d^2 + 0.01}$$

where $f_p$is the promoteclose-fitness, f is the solution fitness and d is the distance to the center point. In this way, fragments located closer to the center point will be granted higher bonus to their fitness than fragments located further. Figure 24 presents locations of 50% most frequently used fragments for regression problem, using classic and promoteclose award methods. It is visible that with the modified method, fragments are located more evenly. This is a result, however, of a purposeful change in the award method and does not need to have a positive impact on overall algorithm efficiency. In fact, the average fitness calculated in the same environment as above is only 0.135, compared to 0.52 achieved by the classic award method.

When comparing fragment distribution charts for regression problem (figures 4-7) with ones for ant and parity problems, one can notice that the fragments are grouped much more for the 2 latter ones than for the former. This may indicate the more modular nature of ant and parity problems but it may as well be a result of the fact that symbolic regression is a real numbers problem while the other 2 are examples of Boolean problems.

8.2.2 Space modularity measures

The fragment location analysis may help in understanding the way GFP works but it does not provide the answer to the question if GFP in fact supports emergence of modularity in problems it solves. The first, basic idea on how to do that is to examine the solutions it produces. Ideally, one could be able to distinguish fragments appearing in various solutions that serve similar functionality, even if their implementation differs. Such fragments might probably be called modules, in particular if they are located closely and if it could be observed that they evolve independently of the rest of solutions.

This is, however, a very difficult task. Brief review of existing work on modularity in GP (see chapter 7) shows that few experiments try to analyse modules appearing in solutions, and if they do, like in Module Acquisition in CGP [13], it is limited to visualising the most commonly performed functionality. In order to automate this process and incorporate information about fragments' locations, in this work a different approach was undertaken.

Gene fragment programming should support emergence of modularity by forming groups of fragments located closely in space, which serve similar functionality and can be used alternatively. Common functionality of fragments within each group might in this way represent a module of the problem. To investigate if such grouping process in fact occurs, a special measure was introduced, *the spatial fragment differentiation, D*. Spatial fragment differentiation shows to what extent the fragments are grouped by similarity of behaviour, or - more precisely - similarity of output in response to the same input. It is defined as follows:

$$D = \frac{\sum_{i=1}^{n} D_i}{n}$$

$$D_i = \frac{\sum_{j=1}^{m} \left(p_i - p_j\right)^2}{m}$$

where m is the number of input data for the problem, n is the number of fragments in the population and p_j are candidate fragments for fragment p_i or, in other words, fragments close to fragment i, as selected by the fragment selection method. Measure D is high if there is little fragment grouping and is low otherwise.

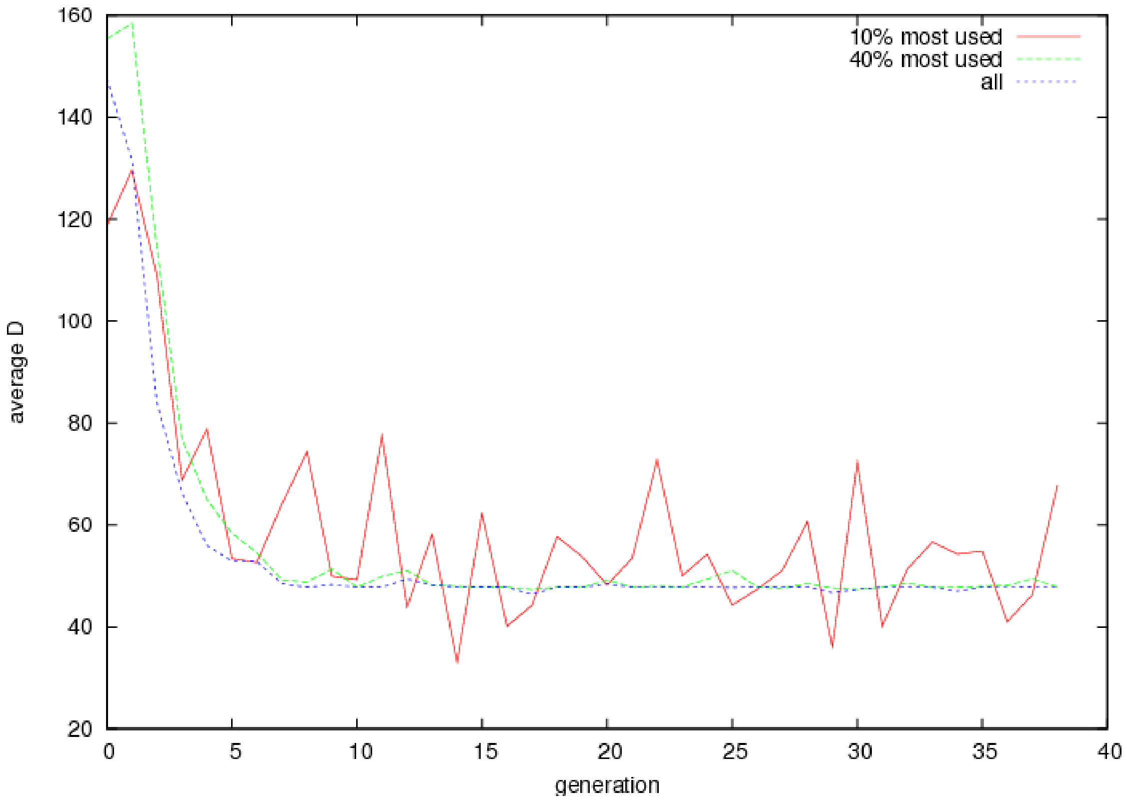


Figure 25: Average spatial fragment differentiation for regression problem

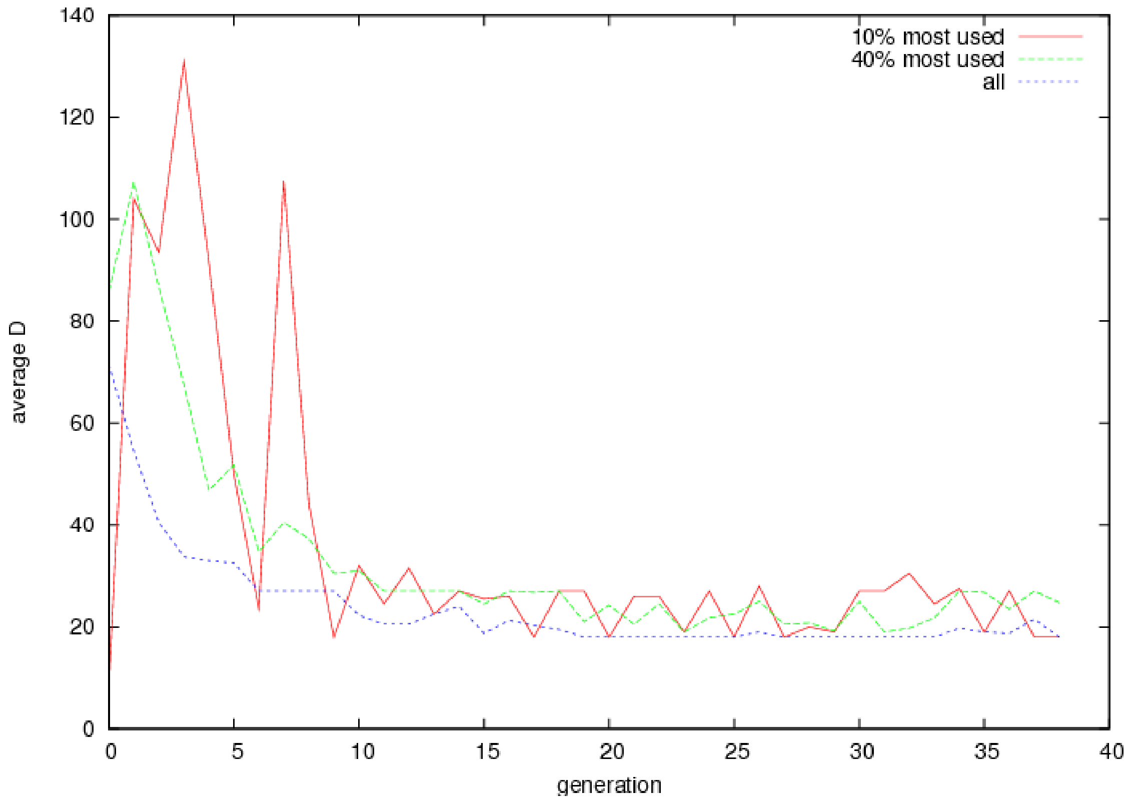


Figure 26: Average spatial fragment differentiation for ant trail problem

Figures 25 and 26 show clearly that the spatial fragment differentiation is decreasing during the evolution. For the regression problem, this measure falls smoothly to around 50 and stays at that level. Fluctuations for the 10% most used fragments are most probably a result of changes caused by evolution, particularly visible because of a smallest number of samples (10% of population). For the ant problem spatial fragment differentiation falls to 20, which is lower than for regression. This may due to the fact that ant trail is a Boolean problem, whereas symbolic regression is not. This seems to be confirmed by rapid but small spikes in this measure for both 10% and 40% most used fragments.

Observations of the spatial fragment differentiation distribution confirms that fragments in GFP are grouped by functionality as a result of evolution. This is a positive fact since such behaviour is one of the principles of GFP.

8.3 Other experiments

8.3.1 Bidirectional vs. one-directional relocation

In chapter 8.2.1 it has been noticed that fragments are generally located in equal distance to the central point of the hyperspace, forming a circle. One reason discussed there was the particular fitness award method, decreasing fitness of fragments located closer to center. Another reason which might be investigated is the

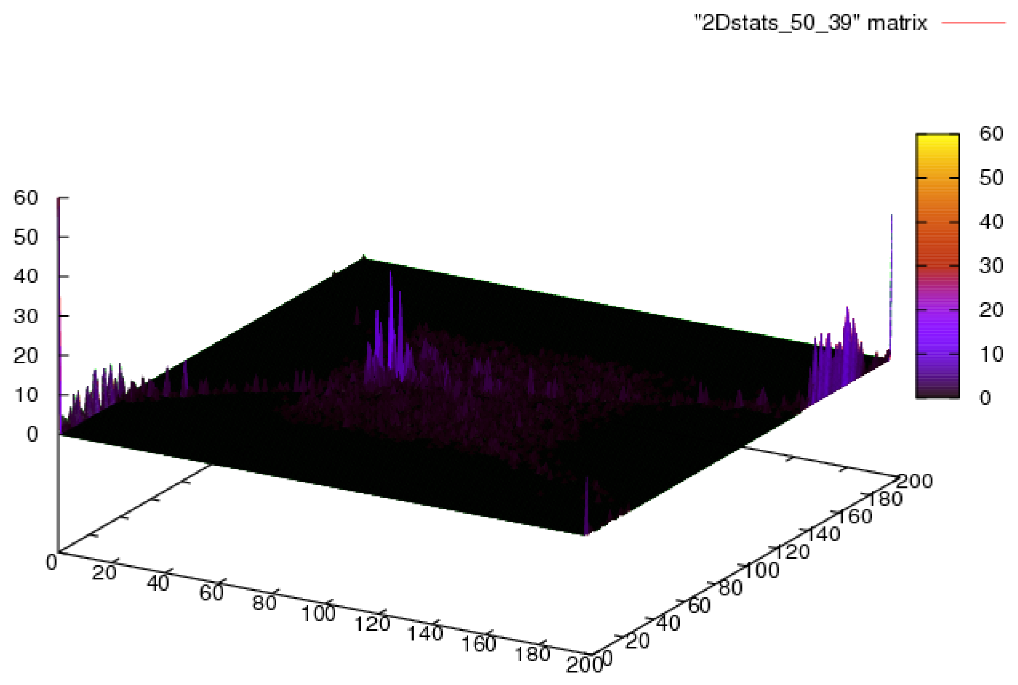


Figure 27: Fragments location distribution for bidirectional relocation

influence of the fragment relocation procedure, which modifies locations of fragments after they are used for constructing a solution. This procedure is described in chapter 4.4.2 and it basically always moves root nodes of the fragments closer to the leaf nodes that they were connected to, the stronger the better this solution was evaluated. The biggest advantage of such procedure is its simplicity - no additional parameters are required. However, it may be interesting to investigate what would be the result of using a different strategy, namely to bring closer fragments used for good solutions and move away those used in bad ones.

The principle issue with such approach is to define the point separating the "good" and "bad" combinations. One idea could be to use median of the previous N evaluations. A quick experiment was done for $N = 63$.

Figure 27 shows the location of fragments for the symbolic regression problem. The distribution is much different than before (compare for example with figure 5), with one group close to the center and other much further. An analysis of how many fragments are used for constructing solutions, in figure 28, reveals that this is at a cost of dramatic decrease of diversity of fragments that are selected. A possible explanation could be that with too many unsuccessful solutions many fragments are moved very far from the center, as a result of what they are never used again. This leaves a very small group of fragments that really matter in the evolution, which quickly leads to stagnation.

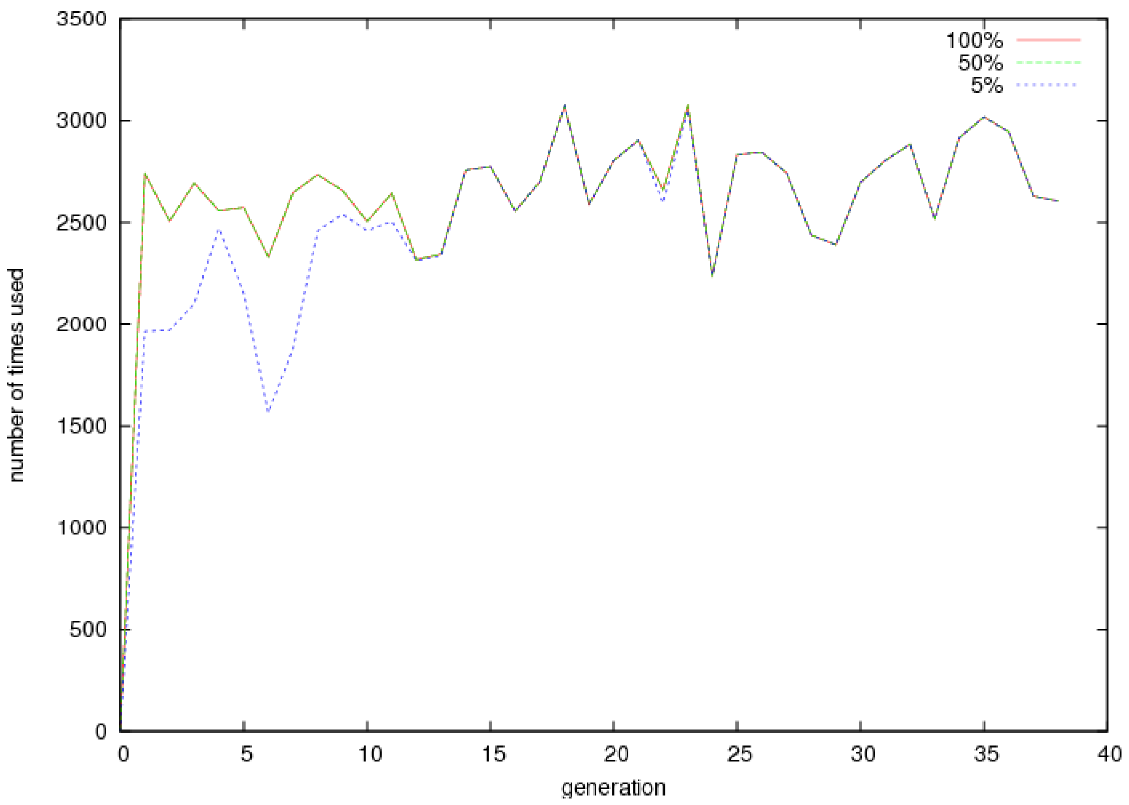


Figure 28: Fragment usage distribution for bidirectional relocation

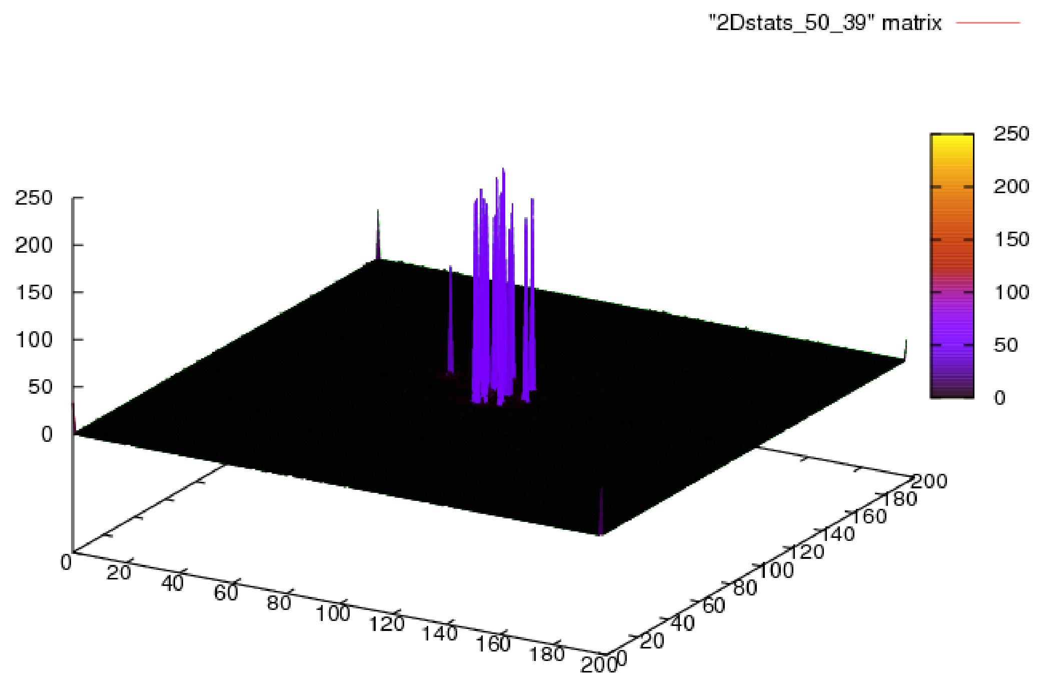


Figure 29: Fragment location distribution when constructing many solutions in each step

8.3.2 Building more solutions at a time

In the basic scheme of GFP, the first fragment of solution is selected in much the same way as the rest, starting the search from the central point. A side effect of that is, however, that the pool of fragments that have a chance to be selected is relatively small (usually 1, 3 or 5). On the other hand, it is unlikely that one fragment will be a good root fragment for many solutions that may be constructed during evolution; in other words - repeatedly choosing a small set of fragments as the first ones for a solution may effectively lead to decreasing their fitness, since most of these solutions may be of poor quality. This is a similar observation to the one in chapter 8.2.1 but the conclusion is different - it may be that the reason for early elimination of fragments close to the central point is not a poorly designed fitness award function, but their overuse.

As a possible remedy for such situation, an experiment was conducted in which a bigger set of fragments was always selected as first ones for solutions. That meant that instead of one solution, many were constructed in each step, each with a different root fragment. An experiment was conducted, using the bidirectional relocation described in previous chapter. The resulting fragment location chart is presented in figure 29.

The fitness chart, in figure 31 is similar to previous configurations of GFP, for example in figure 10, which means there is no or little progress in that matter. As far as fragments' locations are concerned, the observations are not much different from these from the previous chapter - a small set of fragments is located close to the center and practically no other fragments are ever used. That, as can be seen in figure 31, leads to long

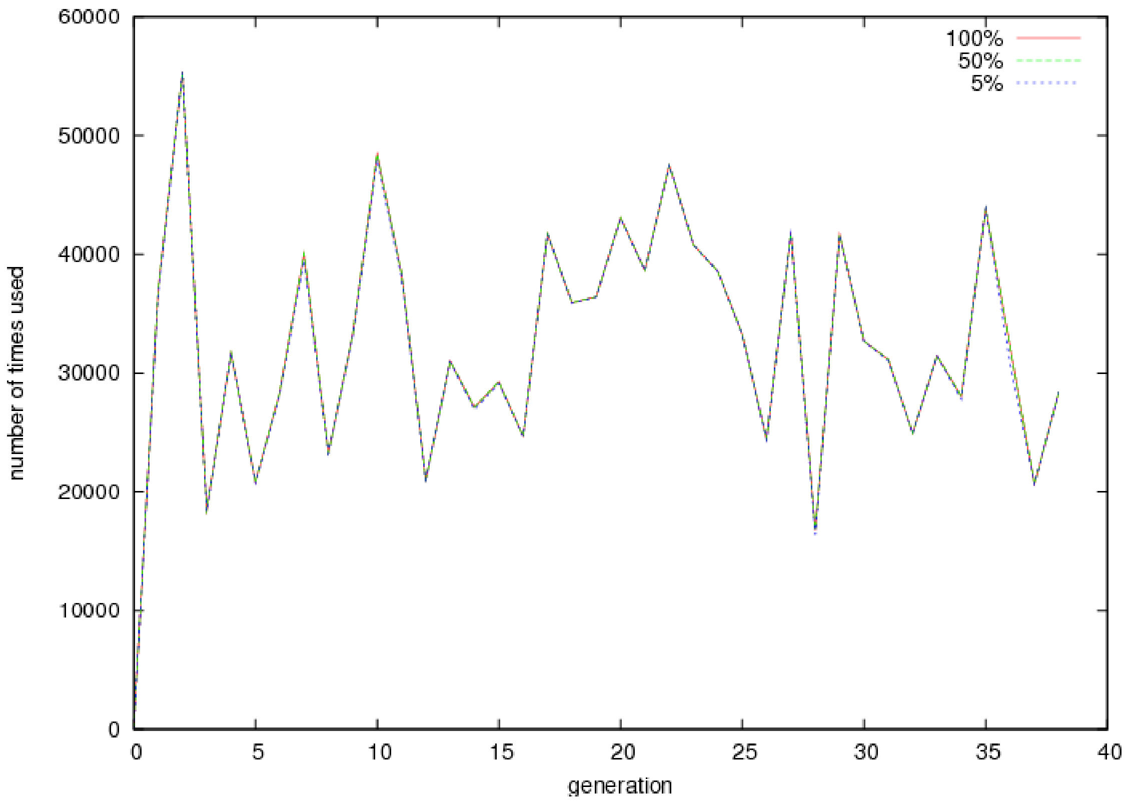


Figure 30: Number of fragments used when constructing many solutions in each step, regression problem

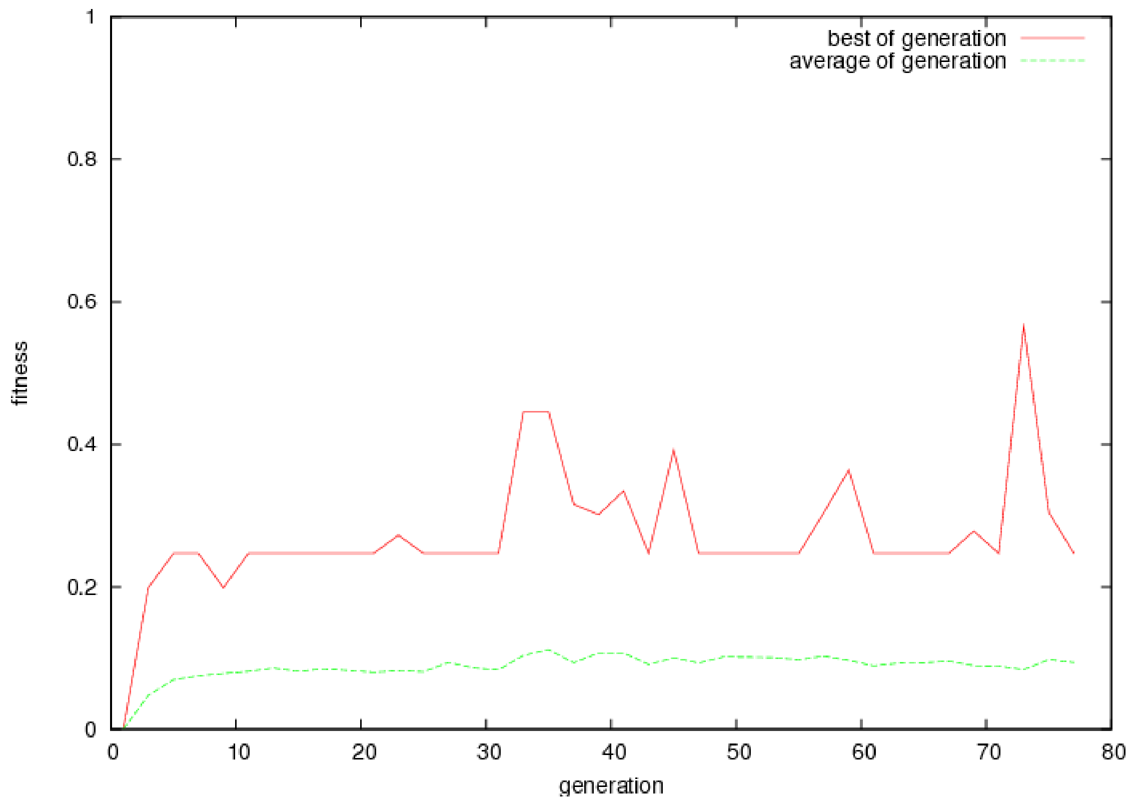


Figure 31: Solutions’ fitness when constructing many solutions in each step, regression problem

periods of stagnation in evolution.

9 Conclusions

This work presented the gene fragment programming (GFP) approach. It is based on evolutionary computing, in particular on genetic programming and to a smaller extent on artificial chemistry. GFP is an innovative approach that addresses the issue of modularity in genetic programming by direct evolution of fragments of solutions, called *gene fragments*, that are subsequently assembled to entities. There is currently no other approach known to the authors that uses a similar concept. Evolving gene fragments presented a set of challenges and design alternatives that have been discussed in this work. To provide a method of storing information about how to connect the fragments, and - moreover - let it evolve as well, each fragment had all its nodes assigned a location, which was a set of coordinates in a virtual hyper-space. In this way, the main goal of evolution was not to directly increase the fitness of each individual in the population, but rather to evolve good combinations of individuals, that together could provide as high fitness as possible. And this new evolution target, in turn, should promote groups of fragments that would improve only in performing a certain functionality - in other words, it should promote the emergence of modularity.

While the general algorithm scheme was defined unambiguously, most steps could be done either in more than way or depending on a set of parameters. In those cases, where it was difficult to assess the best option theoretically, all variants were tested during the experiments at a later stage. The selection of problems on which to test GFP was not a trivial case. None of previous works on modularity contained a clear definition of modularity for GP, and similarly no widely accepted benchmarks for that were found. In the end, 3 problems were selected - the symbolic regression, the ant trail and the even parity problems.

Since gene fragment programming is not a direct modification of a previously developed method, its performance was compared to standard GP. Different parameter combinations were examined and GFP behaviour was measured for each of them. In all cases, the fitness of solutions it created was lower than in classic GP, with less difference for the ant trail problem than for the other problems. Detailed analysis revealed that gene fragment programming did not work in all aspects as expected. For all problems, fragments tended to be located far from the central point, with equal distance to it. As this might have been the reason for smaller effectiveness of GFP, the reasons for that were investigated. Improperly parametrized fitness award function was one of the possibilities and experiments confirmed that certain modifications to it can promote fragments located closer to the center point, though without bringing improvement to produced solutions. Another possibility, a different algorithm that would relocate fragments in both directions after evaluating them, has also been examined. Such modification, on the other hand, lead to moving away a vast majority of fragments, resulting in very high redundancy in the whole population, slowing down evolution without any improvement.

Though quality of solutions they produce is the ultimate goal of most GP algorithms, the motivation behind designing GFP was to support emergence of modularity in problems when evolving programs that should solve them. No metrics for this issue were found in previous works and taking into consideration difficulties in generating dedicated problems, it turned out to be difficult to give a clear answer to that question. On one hand, fragments' locations analyses showed that fragments tend to group themselves in terms of their locations, and dedicated metrics proved that fragments located close to each other tend to be similar in terms of outputs their generate to the same inputs. This is definitely a behaviour that was expected, if GFP was to support detecting modules within problems. Additionally, the grouping behaviour was more clear for the ant trail and parity problems that were expected to have a more modular structure than the symbolic regression problem. On the other hand, more investigations would still be useful to prove that the fragments that are most used in fact represent modules detected in the problem, and are not just side effects of the way GFP works, without any information about the modules of the problems.

Overall, all the goals that were set at the beginning of this work have been achieved. The gene fragment programming approach was designed, implemented and verified experimentally. When assessing its performance a few observations were made which lead to additional modifications to the algorithm. In the end, even though GFP worked well in some matters, it was not possible to definitely confirm all hypothesis that were set at the beginning.

One of the main difficulties when developing GFP was the number of innovative elements that needed to be tuned and verified in practice. This resulted in a rather large set of parameters, controlling various steps of algorithm, without clear knowledge of influence of each of them on others. In future, it might be good to work on a way of testing different elements of GFP separately, before combining all of them together.

A common observation, characteristic for GFP, was also short stability of solutions it generated, visible in strongly fluctuating charts representing fitness and number of elements for best solutions of each generation. Even though there is no clear need for a monotonous fitness chart, low stability is likely to decrease the chances of improving good solutions developed so far. Future versions GFP might need to address this issue.

References

[1] http://bds.ul.ie/libge/libge/santa-fe-ant-trail-problem.html.

[2] http://cs.gmu.edu/ eclab/projects/ecj/.

[3] http://subversion.tigris.org/.

[4] http://trac.edgewall.org/.

[5] http://www.genetic-programming.com/humancompetitive.html.

[6] http://www.geocities.com/tomlahore/santafetrail.jpg.

[7] Peter J. Angeline and Jordan Pollack. Evolutionary module acquisition. In *he Second Annual Conference on Evolutionary Programming*, February 25-26, 1993 February 25-26, 1993 1993.

[8] W. Banzhaf. Genotype-phenotype-mapping and neutral variation: A case study in genetic programming. In *Parallel Problem Solving from Nature III*, 1994.

[9] Dirk Thierens Edwin D. de Jong, Richard A. Watson. A generator for hierarchical problems. In *Gecco*, 2005.

[10] Cândida Ferreira. Gene expression programming: A new adaptive gene expression programming: A new adaptive algorithm for solving problems. *Complex Systems*, 13(2):87–129, 2001.

[11] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley: Reading, MA, 1989.

[12] Thompson A. Harvey, I. Through the labyrinth evolution finds a way: A silicon ridge. In Springer-Verlag, editor, *Proceedings of First International Conference on Evolvable Systems: From Biology to Hardware*, volume 1259, 1996.

[13] Julian Francis Miller James Alfred Walker. Evolution and acquisition of modules in cartesian genetic programming. -.

[14] John R. Koza. Genetic programming: On the programming of computers by means of natural selection. *MIT Press*, 1992.

[15] Eller y Fussell Crane Nicholas Freitag McPhee. A theoretical analysis of the hiff problem. In *Gecco*, 2005.

[16] Jens Ziegler Peter Dittrich and Wolfgang Banzhaf. Artificial chemistries - a review. Technical report, University of Dortmund, 2001.

[17] W. B. Langdon R. Poli and N. F. McPhee. *A Field Guide to Genetic Programming*. http://lulu.com, 2008.

[18] Justinian P. Rosca. Towards automatic discovery of building blocks in genetic programming. Technical report, Computer Science Department, University of Rochester, 1995.

[19] Justinian P. Rosca and Dana H. Ballard. Genetic programming with adaptive representations. Technical report, University of Rochester, Computer Science Department, 1994.

[20] Springer-Verlag, editor. *Cartesian Genetic Programming*, volume 1802. Third European Conference on Genetic Programming, 2000.

[21] R. Keller P. Nordin W. Banzhaf, F. Francone. *Genetic Programming - An Introduction*. Morgan Kaufmann Publishers, Inc., 1998.