

CZECH TECHNICAL UNIVERSITY IN PRAGUE

FACULTY OF ELECTRICAL ENGINEERING

DEPARTMENT OF CYBERNETICS

Bachelor thesis

CASCADE EVOLUTIONARY ALGORITHM

June 2009

Jaroslav Vítků

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Jaroslav Vítků
Study programme: Electrical Engineering and Information Technology
Specialisation: Cybernetics and Measurement
Title of Bachelor Project: Cascade Evolutionary Algorithm

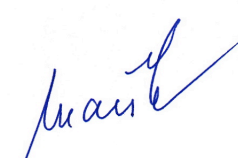
Guidelines:

1. Study conventional models of sequential and parallel evolutionary algorithms (EAs).
2. Propose and implement a cascade EA that exhibits abilities to prevent a premature convergence of the evolved population of candidate solutions.
3. Carry out experiments to assess a performance of the proposed algorithm, statistically evaluate the achieved results and compare the algorithm with traditional sequential and parallel EAs.


Bibliography/Sources: Will be provided by the supervisor.

Bachelor Project Supervisor: Ing. Jiří Kubalík, Ph.D.

Valid until: the end of the winter semester of academic year 2009/2010


prof. Ing. Vladimír Mařík, DrSc.
Head of Department




doc. Ing. Boris Šimák, CSc.
Head

Prague, December 10, 2008

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Jaroslav Vítků
Studijní program: Elektrotechnika a informatika (bakalářský), strukturovaný
Obor: Kybernetika a měření
Název tématu: Kaskádový evoluční algoritmus

Pokyny pro vypracování:

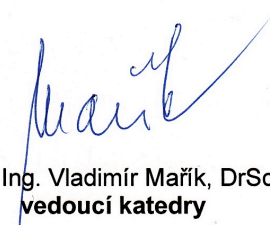
1. Seznamte se se sekvenčními a paralelními modely evolučních algoritmů (EA).
2. Navrhněte a naimplementujte kaskádový EA, který by měl vykazovat schopnost zamezit předčasnou konvergenci vyvíjené populace řešení.
3. Experimentálně ověřte funkčnost navrženého algoritmu, výsledky vyhodnoťte a pokud možno porovnejte s výsledky dosaženými pomocí nejčastěji používaných sekvenčních a paralelních modelů EA.

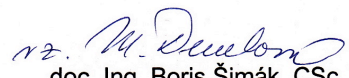
Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí bakalářské práce: Ing. Jiří Kubalík, Ph.D.

Platnost zadání: do konce zimního semestru 2009/2010




prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

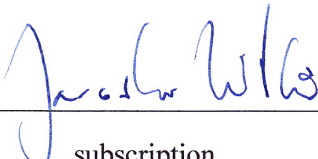

doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 10.12.2008

Declaration

I declare that I have written my bachelor thesis myself and used only the sources (literature, project, SW etc.) listed in the enclosed bibliography.

In Prague 4.6.2009


subscription

Acknowledgement

I would like to thank to everybody who helped me with completing this thesis. Especially to my supervisor Ing. Jiří Kubalík, Ph.D. for his time, significant help and guidance throughout the entire process of development and research.

Abstrakt

Cílem této bakalářské práce je popsat a zhodnotit kaskádový evoluční algoritmus, speciální topologii paralelního evolučního algoritmu, která by měla zabraňovat předčasné konvergenci populace lépe než doposud běžně používané. Tento algoritmus byl navrhnout a naimplementován, experimentálně nakonfigurován a poté byly provedeny testy na několika statických a dynamických problémech. Výsledky byly nakonec porovnány s jednou z klasických topologií paralelního evolučního algoritmu.

Abstract

The aim of this thesis is to propose and review Cascade Evolutionary Algorithm, special case of Parallel Evolutionary Algorithms, which should prevent the premature convergence of population better than the currently used topologies. This algorithm was designed, implemented and experimentally configured, after that some tests on various static and dynamic problems were made. The results were compared with one of the common Parallel Evolutionary Algorithm topologies.

Contents

1 Introduction	7
2 Theory	8
2.1. Evolutionary Computation.....	8
2.2. Basic Terminology	8
2.3. Methods Used in Evolutionary Computation	10
2.3.1 Selection methods.....	10
2.3.2 Crossover	11
2.3.3 Mutation.....	12
2.3.4 Breeding Procedure.....	12
2.4. Simple Genetic Algorithm.....	13
2.5. Spatially Structured Evolutionary Algorithms	14
2.5.1 Motivation.....	14
2.5.2 Description	14
2.5.3 Common Topologies	15
3 Cascade Evolutionary Algorithm	17
3.1. Motivation	17
3.2. Introduction	17
3.3. Description of the Cascade Genetic Algorithm	18
3.4. Implementation	18
3.5. The Principle of Function.....	19
3.5.1 The Island A.....	19
3.5.2 The Rest of Islands.....	21
3.6. Expectations.....	22
4 Experiments	23
4.1. Reference Tasks.....	23
4.1.1 Real functions of n variables.....	23
4.1.2 Dynamic Problems	25
4.1.3 Deceptive Functions	26

4.1.4	Other Binary Problems.....	28
4.2.	Basic Configuration of CGA.....	29
4.3.	Experimental Setting up of CGA main parameters.....	31
4.3.1	Number of Islands.....	31
4.3.2	Number of Migrants	34
4.3.3	Migration Interval.....	35
4.4.	Configured CGA vs. PGA.....	36
4.4.1	CGA configuration.....	36
4.4.2	PGA Configuration.....	37
4.4.3	Tests.....	37
4.5.	Smaller CGA topology vs. PGA.....	44
4.5.1	Smaller CGA Configuration	44
4.5.2	Smaller PGA Configuration.....	44
4.6.	CGA topology for Dynamic Problems.....	45
4.6.1	Small CGA Topology.....	45
4.6.2	Small PGA Topology.....	46
4.7.	Discussion	48
4.8.	Conclusion.....	49
Attachments		51
Used Abbreviations		68
Bibliography		69

List of Tables

Table 4.2.a – Basic Configuration of CGA topology	29
Table 4.2.b – Basic configuration of the problem.....	29
Table 4.2.c – Basic configuration of common island	30
Table 4.2.d – Basic Configuration of the island A	30
Table 4.3.a – Number of Islands - 4.....	31
Table 4.3.b – Number of Islands - 6	33
Table 4.3.c – Nmber of islands - 8	33
Table 4.3.d – Number of Islands - 10.....	33
Table 4.3.e – Number of Migrants - 5	34
Table 4.3.f – Number of Migrants - 10.....	34
Table 4.3.g – Number of Migrants - 12.....	34
Table 4.3.h – Number of Migrants - 25.....	34
Table 4.3.i – Number of Migrants - 40	34
Table 4.3.j – Number of Migrants - 45	35
Table 4.3.k – Migration Interval - 10.....	35
Table 4.3.l – Migration Interval - 20	35
Table 4.3.m – Migration Interval - 50	35
Table 4.4.a – The Configured CGA topology	36
Table 4.4.b – Configuration of PGA topology.....	37
Table 4.4.c – Real Function Configuration - Schwefel.....	38
Table 4.4.d – CGA vs. PGA - DF3.....	38
Table 4.4.e – CGA vs. PGA - HIFF	39
Table 4.4.f – CGA vs. PGA – Schwefel’s function	39
Table 4.4.g – Additional Configuration table for the Ošmera’s problem	39
Table 4.5.a – Smaller CGA Configuration	44
Table 4.5.b - Smaller PGA configuration.....	44
Table 4.5.c - DF3 task solved by Smaller Topologies	45
Table 4.5.d - HIFF task solved by Smaller Topologies	45

Table 4.6.a – Small CGA for Dynamic Problems conf.....	45
Table 4.6.b – Small PGA Topology for Dynamic Problems conf.	46

List of Figures

Figure 2.3.a – Roulette selection principle.....	11
Figure 2.3.b – One point crossover.....	12
Figure 2.3.c – Example of mutation.....	12
Figure 2.3.d – Breeding procedure	13
Figure 2.4.a – Generational evolution model.....	13
Figure 2.4.b – SGA generational procedure.....	14
Figure 2.5.a – Mesh graph.....	16
Figure 2.5.b – Star graph	16
Figure 2.5.c – Ring graph.....	16
Figure 2.5.d – Grid graph.....	16
Figure 3.3.a – CGA topology example.....	18
Figure 3.5.a – Custom Evolution Model	20
Figure 3.5.b – Common Evolution Model.....	21
Figure 3.6.a – Expected Behaviour of CGA.....	22
Figure 4.1.a – Sphere (De Jong’s function)	23
Figure 4.1.b – Rosenbrock’s valley	24
Figure 4.1.c – Rastrigin’s function	24
Figure 4.1.d – Schwefel’s function.....	25
Figure 4.1.e – Ošmera’s Function.....	26
Figure 4.1.f – Deceptive Function 3.....	27
Figure 4.1.g – Partially Deceptive Function.....	27
Figure 4.1.h – Royal Road problem.....	28
Figure 4.1.i - Hierarchical If and Only If problem.....	28
Figure 4.3.a – DF3 task solved by 4-island CGA (typical run)	32
Figure 4.4.a – CGA configuration	36
Figure 4.4.b – PGA configuration.....	37
Figure 4.4.c – Ošmera’s function solved by the CGA (typical run)	40
Figure 4.4.d – Ošmera’s Function solved by the PGA (typical run).....	41

Figure 4.4.e - absolute error from ideal solution on CGA topology (average values).....	42
Figure 4.4.f - absolute error from ideal solution on PGA topology (average values).....	43
Figure 4.6.a - Ošmera's Function Solved by the Small CGA Topology (typical run)	46
Figure 4.6.b – Ošmera's Function Solved by the Small PGA Topology (typical run)	47
Figure 4.6.c - CGA - absolute error of the best solution found (average values)	47
Figure 4.6.d - PGA - absolute error of best solution found (average values)	48

Chapter 1

Introduction

The aim of this bachelor thesis is to describe the function of Cascade Evolutionary Algorithm (CEA), special topology of Parallel Genetic Algorithms. This custom topology should prevent the premature convergence of population better than the currently used ones. The main feature of this algorithm is sequencing the islands in a linear topology, where the first island generates random solutions and the last island is meant to be an output of the topology. In this thesis the Cascade Genetic Algorithm (CGA) will be implemented, configured and tested on subset of common reference tasks. The results will be compared with similar configured common island genetic algorithm topology. As I will explain in the following chapter, the Genetic Algorithm (GA) is special case of Evolutionary Algorithms (EA). For simplicity, the functionality of the CEA will be presented using experiments performed on CGA.

At the beginning I will describe the basic terminology and principles of Evolutionary Computation (EC) used in this work. After that I will try to explain purpose, principles and advantages of spatially distributed evolution. At the end of chapter I will mention some commonly used island topologies.

The following chapter will contain the complete description of the Cascade Genetic Algorithm. Here I will describe the purpose of this new approach and the principle of function. This algorithm partially uses custom evolution models, so the implementation and the principle of these models are also described here. At the end of chapter the expected behaviour of this topology will be presented.

The chapter 4 describes the performed experiments used for comparison of algorithms. At the beginning there are mentioned the main tasks commonly used for testing of Genetic Algorithms. Then the complete description of tested algorithms and their configuration follows. The Cascade Genetic Algorithm is here experimentally set up and compared to the similar configuration of common Parallel Genetic Algorithm topology. For this comparison are used some of the common static and dynamic tasks. The best solutions found by the algorithms are presented in tables and charts.

At the end of this thesis I will summarize the results of performed experiments. By comparing the expectations with the results I will decide whether they were correct or not. I will recapitulate the behaviour and features of this algorithm and mention some potential advantages of this new approach against the commonly used ones.

Chapter 2

Theory

2.1. Evolutionary Computation

The Evolutionary Computation is relatively new part of artificial intelligence; concretely it belongs into nature inspired computation. This method of computing is inspired by Darwin's theories about a natural selection and evolution, where (in some way) those good individuals survive and transfer their qualities into offspring, by this way the entire species are becoming stronger in the environment. The same principle can be relatively easily and effectively used in computer technology. We can do this by coding one entire solution of our problem as an *individual*. And what do we need to know about each individual in the nature? We need to know his properties, and how strong he is. The properties are usually coded in individual's genome, so the *genome* will represent the solution of our problem. The second thing we need to know will be represented by *fitness* - some function value defining how strong the individual is, i.e. how good is the solution that he represents.

As well as Darwin's natural selection, the Evolutionary Computation is stochastic and very robust approach to solve given tasks. The main difference from classical approaches is that the evolution is not operating with just one solution, but with entire *population* of solutions (*individuals*). From this property we can expect that this method can't be as fast as the classical ones and we need more memory too. But the EC has many advantages. The main advantage is that it is very robust and able to find the solutions mainly unreachable by the conventional methods. And the principle of EC is relatively simple but very effective.

The power of Evolutionary Computation is described by the following sentence very well: J. Holland: "It's best used in areas where you don't really have a good idea what the solution might be. And it often surprises you with what you come up with." [3], [4].

2.2. Basic Terminology

The evolution strategies can be divided into several main categories:

- Genetic Algorithms (GA)
- Genetic Programming (GP)
- Learning Classifiers Systems (LCS)
- Evolution Strategies (ES)
- Evolutionary Programming (EP)

As the EC use an analogy of natural evolution, they has similar terminology with natural genetics, so here are some of the most important terms:

- **Genome:**
 - unambiguous representation of one solution
 - gene sequencing of constant length
 - in the ideal case all of the genome combinations should be acceptable
- **Gene:**
 - elementary units from which chromosomes are made
 - each of genes is located at its place called locus (allele)
 - it is equal to one bit in many applications
- **Fitness:**
 - definition of quality measure assigned to genome
 - computing of fitness function cannot be too costly
- **Individual:**
 - the entire representation of one solution
 - contains the genome and a value of the fitness function
- **Population:**
 - defined set of individuals used for computation
- **Reproduction:**
 - creating offspring from parents by combining their genomes in some manner
 - used for creating new individuals in the population
 - also called breeding
- **Crossover:**
 - method of combining parents to produce (one or) two offsprings
 - for example we can use one or two point crossover
- **Mutation:**
 - random small modification of newly generated offspring
 - used for avoiding premature convergence in the population
- **Island:**
 - spatially separated part of the Evolutionary Computation
 - by default one island holds one its own population
- **Migration:**
 - transfer of selected amount of individuals from one island to another
 - usually the strongest ones are selected for the migration
- **Evolution model:**
 - defined method for creating new population from the old one
 - two main possibilities:
 - steady-state
 - generational (I use only generational one here)
- **Genotype:**
 - information stored in genome (the genome of animal)

- **Phenotype:**
 - o what the information in genome means decoded (animal itself)
- **Elitism:**
 - o the size of elitism specifies how many of the best individuals from the actual population will be copied into the new generation without any modification
 - o this means that defined number of best individuals won't be lost

The other terms used here are written at the end of this thesis, for more detailed explanation you can read some available literature.

In my thesis I use Genetic Algorithms, which are characterized by representing individuals as consecutions of characters chosen from a commonly finite alphabet [4]. The alphabet can be composed only from '0' and '1' in the case of bit-vector genome.

2.3. Methods Used in Evolutionary Computation

In the EC there are some common methods and approaches used for selecting individuals, their combination, mutation etc. I will describe here only the most important ones.

2.3.1 *Selection methods*

At first we need some method for selecting individuals, selection is done for several purposes. First, the most important is selecting the individuals for their reproduction. This selection method should be stochastic, but also based on the fitness value. These two attributes should prefer selection of better individuals, but also give a chance to the bad ones, which is very important for keeping the population diverse. Another kind of selection is selecting of best/worst individuals in the population, that is used for selecting elites, selecting the individuals for migration, and selecting the worst individuals to be replaced by newly incoming ones from the others islands. (In case that we are using some kind of Parallel Genetic Algorithm.)

Tournament Selection

The most frequently used selection method is called Tournament Selection, which connects both of the mentioned conditions and the selection is based on the fitness value and a random selection together.

For example if the tournament selection has specified size 3:

1. Three individuals are chosen from the population randomly.
2. From these three is chosen the best one.

The important property is that when using this selection method, every individual has a chance to be selected (reproduced).

Roulette wheel selection

This is simpler, but not so good selection method, which also prefers best individuals to the worse ones. The individuals own the place directly proportional to its fitness:

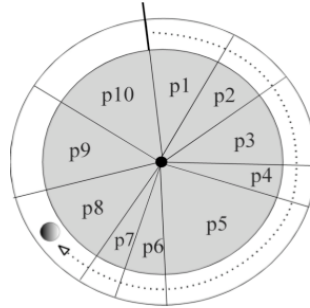


Figure 2.3.a – Roulette selection principle

[3]

After that the random position on the wheel is chosen and so the probability of selection of each individual is:

$$P_i = \frac{f_i}{\sum_{j=1}^{PopSize} f_j}$$

Where f_i is the fitness value of i-th individual, and P_i is the probability of selection. This method is also known as *fitness proportionate selection*. But this selection method is not suitable when there are big differences in the fitness values; it also has comparatively big error.

Best selection

Simply selects a defined number of the best/worst individuals from the population.

2.3.2 Crossover

When having two selected parents, we need to combine them to produce new offsprings having some (hopefully) better properties. If the one-point crossover is selected, we simply get two genomes of parents and partitione them in randomly selected location. Then the new offspring is composed from two parts, each taken from different parent.

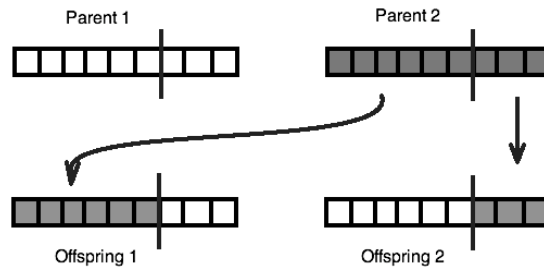


Figure 2.3.b – One point crossover

The purpose of using crossover is exploration of state space. By combining two good individuals, the GA is trying to generate new and better individual, which provides good direction of exploration.

2.3.3 Mutation

A mutation has the same purpose as a mutation in the natural selection. This method prevents the population from the premature convergence; it enables the algorithm to be more robust. Thanks to this the algorithm is less susceptible to get stuck in a local optima.

The principle is that each of the genes is mutated with some small probability. In the GA this means change of character. In case of bit-vector genome, the mutation means usually simple bit flipping.

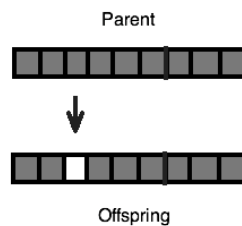


Figure 2.3.c – Example of mutation

2.3.4 Breeding Procedure

The mutation is usually applied to the offspring just produced by the crossover, the crossover is applied at two individuals chosen using some of the selection methods, so the breeding procedure has the following form:

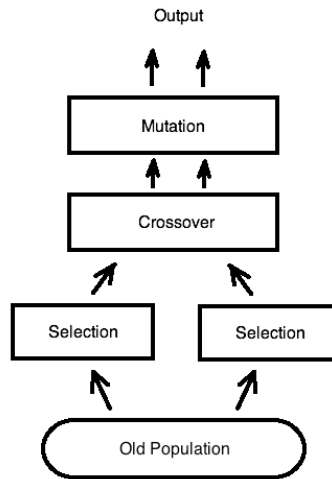


Figure 2.3.d – Breeding procedure

2.4. Simple Genetic Algorithm

The simplest way to start with the Evolution Computation is a Simple Genetic Algorithm (SGA). It contains one population holding some constant amount of individuals. The individuals are often composed of bit-vector genome (with no constraints) and simple (single-multiobjective) fitness value stored for example as a floating-point variable. In case when we are using generational evolution model, the algorithm is as follows: pick individuals from the old population, breed them (i.e. create offsprings), evaluate and place into the new population. This is repeated until some stopping criterion is reached.

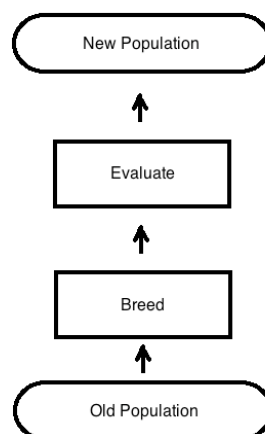


Figure 2.4.a – Generational evolution model

Simple generational Genetic Algorithm procedure:

1. Choose initial population
2. Evaluate the fitness of each individual in the population
3. Repeat until termination: (time limit or sufficient fitness achieved)
 1. Select best-ranking individuals to reproduce
 2. Breed new generation through crossover and/or mutation and give birth to offspring
 3. Evaluate the individual fitness of the population
 4. Replace worst ranked part of population with offspring

Figure 2.4.b – SGA generational procedure

[1]

Note: typical number of newly produced individuals is: population size - elitism

2.5. Spatially Structured Evolutionary Algorithms

2.5.1 *Motivation*

In the real nature, there are many separating factors between the individuals, for example rivers, forests etc. The *panmictic population* is one where all individuals are potential partners. This assumes that there are no mating restrictions, either genetic or behavioural, upon the population, and that therefore all recombination is possible. [9]

The common approach to simple, one-island, Evolutionary Computation uses a *panmictic population*.

But Darwin realized long ago that the population should have a spatial structure, which has an influence on population dynamics. For instance he remarked how, when they were isolated on islands, some species evolved differently from others that lived in more open environments. [7]

So if the separating factor is removed and the individuals can „meet“ freely each other, the population become more uniform, with a decrease in genetic diversity.

2.5.2 *Description*

In order to improve the functionality of EAs, there were evolved Spatially Structured Evolutionary Algorithms. These algorithms have the same functionality as the EAs, with the difference that their population is divided into a number of separated parts. Each part is called an *Island*. On each island there is one population containing some constant amount of individuals. As in the nature, the strongest individuals are usually capable of migration between particular islands. Being partially separated, individuals

on each island evolve in slightly different ways. So this feature partially eliminates the flaw of classical EAs using panmictic population - premature convergence of population.

Here are the main differences between classical EAs and this improvement:

- One population is said to be palced on an *Island*.
- There is defined several islands in a topology.
- The *topology* defines how the particular islands are interconnected.
 - That means that for each island is defined sevral target islands where to send the chosen migrants.
 - Each island can have zero, or an arbitrary number of target islands.
- There is specified *migration interval* defining how many generations to wait between sending chosen individuals to other islands.
 - When running synchronous island topology, there is usually defined only one migration interval, all of the islands are allowed to run only as fast as the slowest one, so the migration starts on all of the islands at the same time.
 - When running asynchronous island topology, each island has its own migration interval, so that the migration starts at any time.

So on each of the islands resides one panmictic population, which sends some number of its best individuals to some other islands.

Again, the special case of Spatially Structured EAs is Island model PGA, which I use for comparison with CGA. Compared to comon GAs, Parallel Genetic Algorithms proved many advantages, for example the solution is found faster in many cases, the second and more important advantage is that these algorithms have bigger overall population diversity, so they are even more robust. For more information i.e. [5].

2.5.3 *Common Topologies*

There are some commonly used topologies in PGAs, each of them has some specific properties, I will briefly describe some of them here.

A complete graph

It is also called *Mesh graph* or *fully interconnected*, in this case there is every island connected to each other. This topology affords the fastest propagation of best individual to the other islands. We can need this when solving some dynamic problems. But the number of migrants shouldn't be too big, for keeping up some separation between particular populations. The edges in this graph are bidirectional.

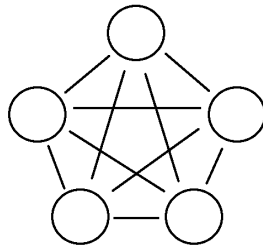


Figure 2.5.a – Mesh graph

A star graph

The other topology has one center and other leaf islands.

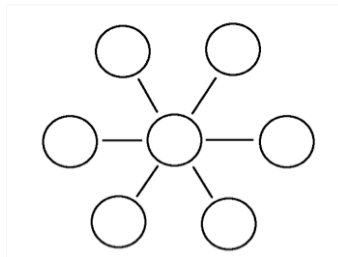


Figure 2.5.b – Star graph

Ring graph

The islands are interconnected to the ring topology. The edges of this graph can be one, or bidirectional. When we need faster propagation of individuals between islands we use bidirectional edges.

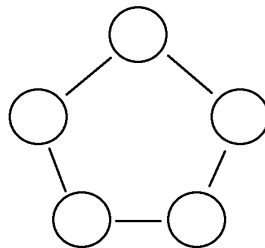


Figure 2.5.c – Ring graph

Grid graph

This graph has usually bidirectional edges.

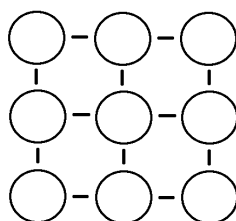


Figure 2.5.d – Grid graph

Chapter 3

Cascade Evolutionary Algorithm

3.1. Motivation

Despite its qualities the Parallel Evolutionary Algorithms suffers from unwanted premature convergence of population. Because of this they are unable to escape from a potential very strong local optima and thus to reach some solutions of given task. Regardless of the circumstances, all of the populations in the common PEA always converge to some optimum. The Cascade Evolutionary Algorithm, proposed in this thesis, is trying to fix this partial flaw by injecting new, randomly generated, individuals into the custom EA island topology and thus maintain the population diversity.

Again, the functionality and principles of CEA will be presented here on the CGA, the special case of EAs. At first the principle of CGA will be described, then a several experiments will be performed in order to configure single parts of the algorithm, such as a number of islands in the topology, a migration frequency and a number of migrants. After that the CGA will be configured using a results from the previous experiments. This configured CGA will be compared to some similar PGA with islands connected to one directional ring topology, which will represent the current PGA configurations. Finally these results will be summarized and I will try to decide whether this new approach has better properties than the standard ones.

3.2. Introduction

The idea of such algorithm is preventing of the premature convergence in the entire island topology by attaching an island used for permanent generating of random individuals and injecting them into the topology. This feature itself can't be much contributive, because the new individuals have significantly worse fitness than the current ones. These new immigrants would have only very small chance to stay in the population with high average fitness. A particular solution could be in increasing the number of immigrants, but it decays the average fitness on a target island. The other solution is in sending some improvement of random individuals, which we can generate and after that enhance for example using another evolutionary run – on another island. So the best solution here seems to be in attaching some several islands into a queue. Let's say that on the left side there is some kind of input and the island on the right side is the output of entire topology.

3.3. Description of the Cascade Genetic Algorithm

The main features of CGA are:

- Sequencing the islands in a linear topology with islands connected by one directional links.
- Introduction of one island attached at the beginning of this topology, which permanently generates new random solutions.

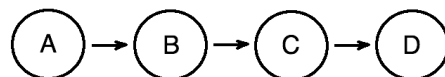


Figure 3.3.a – CGA topology example

In this linear topology there is an island A meant to be a generator of random solutions, and an island D is proposed as an output evolving the best individuals in the topology. All of the islands, from second to the last one, combine their evolved solutions with newly incoming individuals from the predecessor populations. As going along the topology, divergent populations with higher and higher average fitness values are expected.

So the CGAs principle has the following form:

- Island A generates random individuals and sends them onto the next island
- As going through the topology, individuals are improved by the evolution
- The last island receives entirely new, but evolved, genetical material

I hope that the main advantage of this approach is a fact, that into the last island is injected absolutely new genetic material all the time of evolutionary run, so the topology should hold the necessary divergence in the population much longer than the common ones. For example in the Mesh topology where all of the islands are interconnected, there occurs partial waste of migration bandwidth and a loss of the population diversity. When some of the islands finds better solution than the others and send it to them, the descendants of this migrated individual may occupy the whole population on the target islands, so that the migration is not effective anymore.

3.4. Implementation

Experiments and the algorithms were implemented and tested in a programming language Java 1.5. In the hope to save some work the *ECJ 18: A Java-based Evolutionary Computation Research System* was used, which is very sophisticated Java library containing various functions. For real-time graph plotting was used *SGT: Scientific*

Graphics Toolkit. For post-processing of the experiments was used well known *Matlab*. [15] [16] For more information about the implementation, please check the *Attachment*.

3.5. The Principle of Function

As explained in the previous chapter, the CGA has two or more islands, where the first island (I will call it island A) generates new individuals. The following islands should be evolving these individuals to higher level. Finally, on the last island (island D) there are the best results expected. In this island should be the biggest selection pressure, and the population should contain highly evolved individuals with good average fitness. In this chapter I will describe the CGA topology containing four islands for simplicity.

3.5.1 *The Island A*

The island A sends newly generated individuals on the island B once after a certain number of generations. Island A sends the entire population, so that all of the individuals on the island B are replaced.

In order to achieve better results, the island A generates completely random individuals only after sending its entire population to the island B. In the remaining time only generates new random individuals, and if they are better than some in the current population, they are replaced. So even the island A itself generates random, but relatively strong, individuals which are then more evolved by the following islands. The rest of topology has similar parameters to the classical PGAs.

So the function of the island A looks like as follows:

1. check whether the entire population should be sent:
 - if **no**:
 - randomly generate new individuals (the number of generated individuals is equal to the size of population on the island)
 - for each new individual:
 - evaluate them
 - check if it has better fitness value than the worst one in the actual population
 - if yes: replace them
 - if no: throw it away
 - if **yes**:
 - send the entire population
 - generate the entire population randomly
2. go to 1.

For this purpose in the ECJ there was implemented custom Evolution Model (EM), composed from *breeding sources* and *breeding pipelines*.

The breeding source is equal to some selection method, and the breeding pipeline is something which modifies the genome or an individual in some manner (e.g. mutation, crossover or whatever..)

This EM nicely describes the function of the island A:

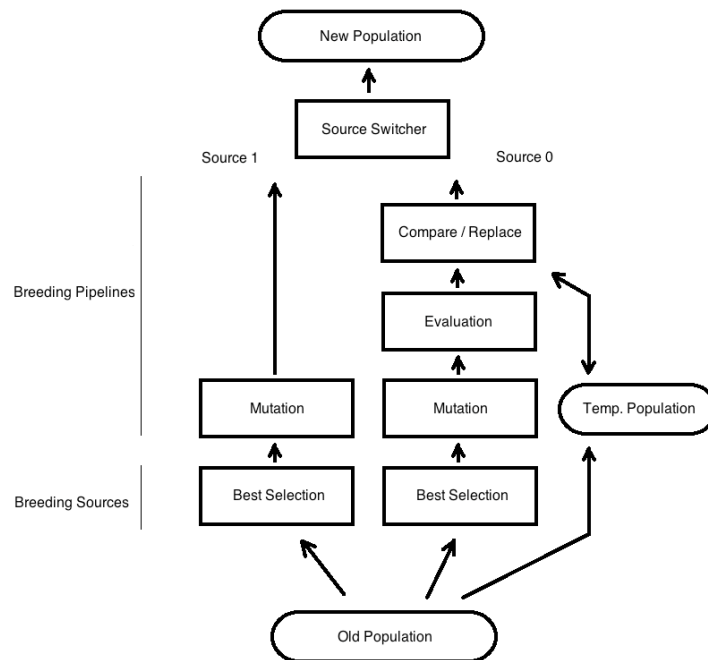


Figure 3.5.a – Custom Evolution Model

Best Selection:

- Consecutively selects and puts forward the entire sorted population (in this case from the best to the worst one).

Mutation:

- Mutates the individual on the input with given probability and sends to the output.
- The mutation probability is the probability of bit-flipping for each of genes.
- In this case $P_{mut} = 0.5$, so the mutation is equal to a random generating.

Evaluation:

- Evaluates the individual for the comparison purposes.

Compare/Replace:

- If the first individual was received (in the actual generation), copy the old population into a temporary place.
- Compare the individual on the input with the temporary population and replace if it is better than some of the current ones.

Source Switcher:

- This block holds as the breeding source the source 0.
- Once after the specified number of generations it switches the actual breeding source to the source number 1, at the end of generation switches back to the source 0.

3.5.2 *The Rest of Islands*

The rest of the islands in the CGA topology have a common functionality and a commonly used EM, which is also default in the ECJ. So it looks as follows:

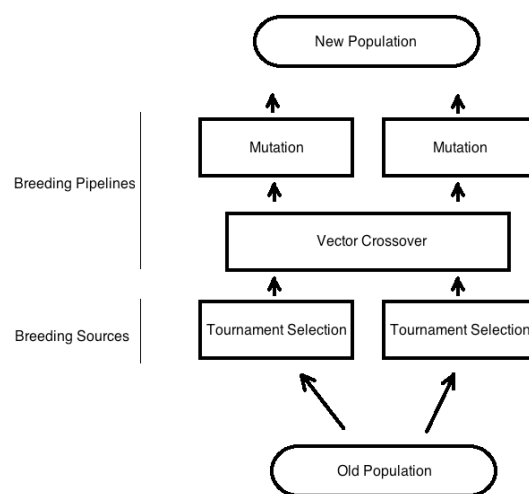


Figure 3.5.b – Common Evolution Model

So on these islands there is running a classical GA. The only differences are that on the island B there is entirely new population in some defined time period and the island D is on the end of the topology, so it doesn't send migrants to any other island. In this EM, there is used a tournament selection, a crossover and the mutation with some very small probability of course.

The migration is defined in this manner:

- Select defined number of best individuals on the island.
- Send them to the target island (the chosen individuals stay on the source island also).
- On the target island: select the corresponding number of the worst individuals, and replace them by newly incoming immigrants.

Note: In the ECJ, there is the evaluation process is defined on some another place.

3.6. Expectations

From the previously mentioned facts and hypotheses we can expect following behaviour of this algorithm. In case of using the synchronous island model and solving some static and difficult problem, the graphs of average fitness should look similar to this one:

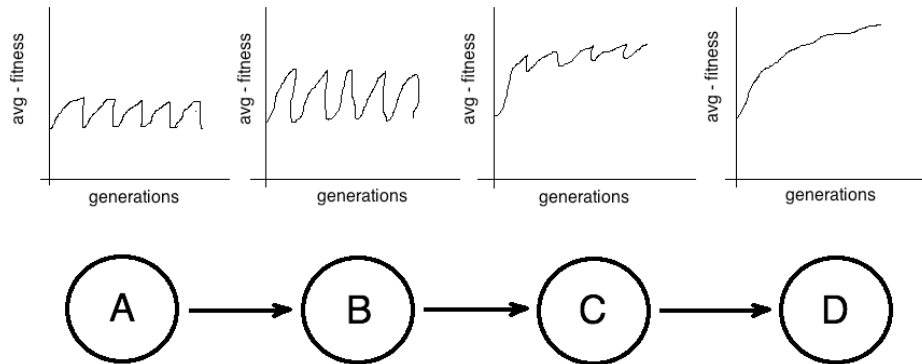


Figure 3.6.a – Expected Behaviour of CGA

Each of the graphs represents the island drawn below. From the cogs on the average fitness we can see the migration intervals and also the reason why we should connect several islands into a queue. Finally, a new genetical material supplies the island D, so this island is able to find still better and better solutions.

I suppose that too short topology may not find the ideal solution, because of decaying the average fitness by the island A. In the other hand, too long topology will require unnecessarily much processor time and will be also too slow for dynamic problems.

Chapter 4

Experiments

4.1. Reference Tasks

For configuring and quality comparison of Genetic Algorithms there are used some common reference tasks. Here are several kinds of these functions:

- Real functions of n variables
- Dynamic problems
- Deceptive functions
- Other binary problems

I will describe those I have used for testing purposes.

4.1.1 *Real functions of n variables*

These functions are defined as one or more dimensional real continuous functions with one global and usually many local strong extremes. They are mainly used for testing exploration and a speed of GAs. For more information and other functions: [10]. Here are examples of these functions in the case of two input variables:

Sphere function

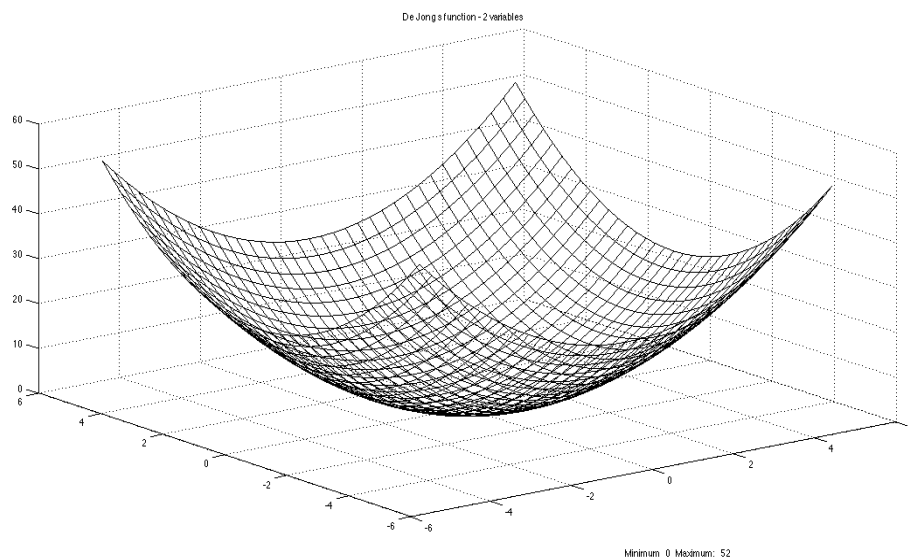


Figure 4.1.a – Sphere (De Jong's function)

$$f(x) = \sum_{i=1}^n x_i^2 \quad -5.12 \leq x_i \leq 5.12 \quad i = 1:n \quad \text{minimum: } f(x) = 0 \quad x(i) = 0 \quad i = 1:n$$

Rosenbrock's valley

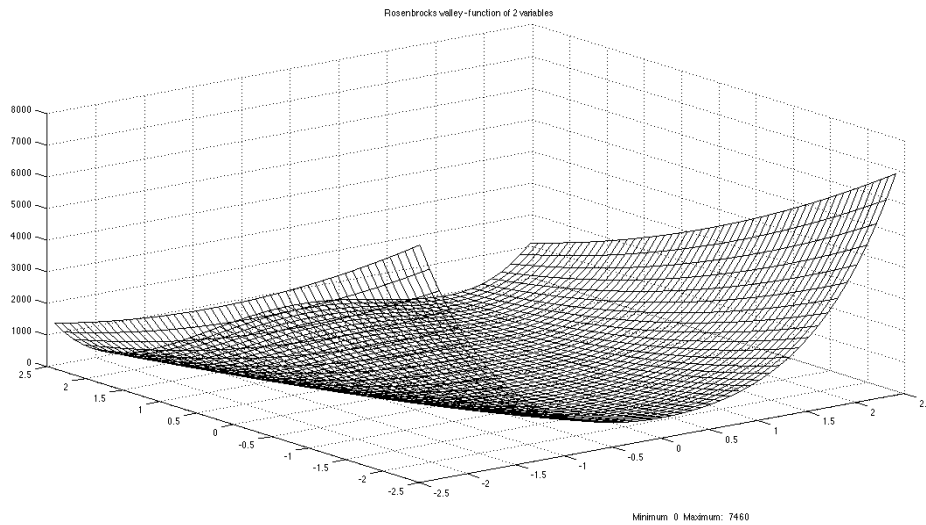


Figure 4.1.b – Rosenbrock's valley

$$f(x) = \sum_{i=1}^{x-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \quad -2.048 \leq x_i \leq 2.048 \quad i = 1 : n - 1$$

minimum: $f(x) = 0 \quad x(i) = 1 \quad i = 1 : n$

Rastrigin's function

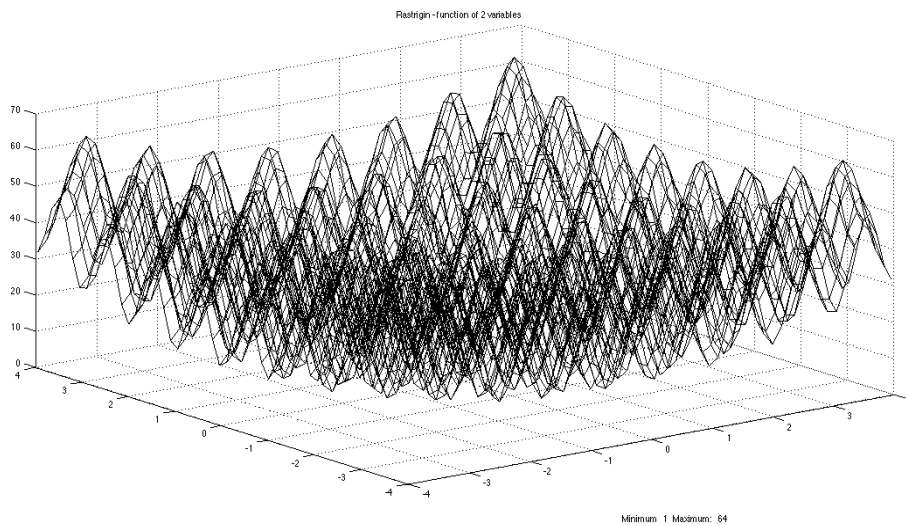


Figure 4.1.c – Rastrigin's function

$$f(x) = 10n + \sum_{i=1}^x (x_i^2 - 10 \cos(2\pi x_i)) \quad -5.12 \leq x_i \leq 5.12 \quad i = 1 : n$$

minimum: $f(x) = 0 \quad x(i) = 0 \quad i = 1 : n$

Schwefel's function

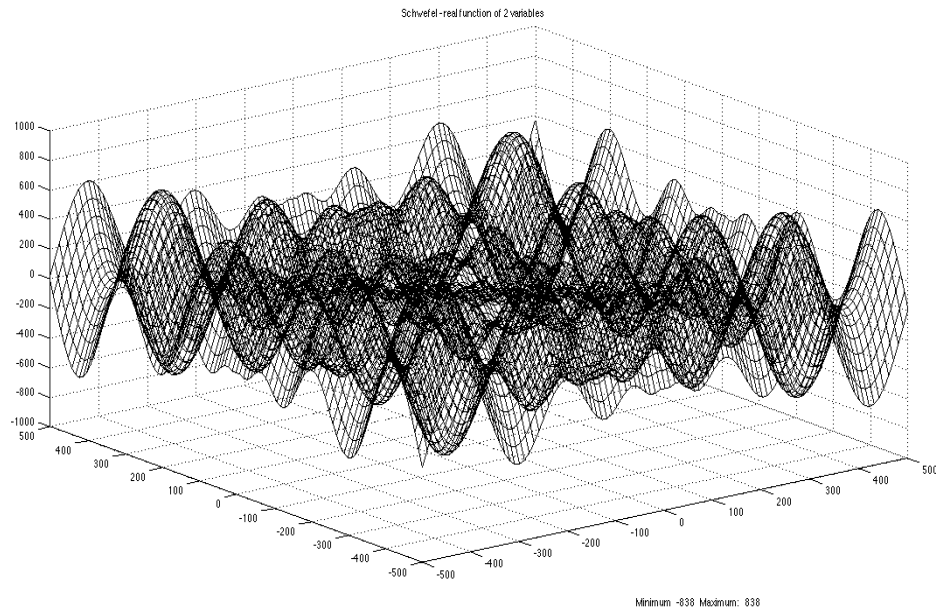


Figure 4.1.d – Schwefel's function

$$f(x) = \sum_{i=1}^n -x_i \sin \sqrt{|x_i|} \quad -500 \leq x_i \leq 500 \quad i = 1 : n$$

$$\text{minimum: } f(x) = -n \cdot 418.9829 \quad x(i) = 420.9687 \quad i = 1 : n$$

4.1.2 Dynamic Problems

Oscilating Knapsack Problem

This problem represents the situation when we have a small knapsack and many things to pack into it. Each of the things has its own value of interest and weight defined. The knapsack has some given maximum load. An actual set of things in the knapsack is coded into the genome. In the Oscilating Knapsack Problem there are two maximum loads defined which are swapped after some defined number of generations. In this simplified case the fitness value of each individual equals to a sum of its genes identical to the genes of the ideal solution.

This problem is used for testing the reaction time of GAs to changes of the target solution.

Ošmera's function

Ošmera's function is another dynamic problem. This function has nonmonotonic changes, which is also very important for assessing the properties of the GAs. The change appears every 20 generations. Here are the equations:

$$g_1(x,t) = 1 - e^{-200(x-c(t))^2} \quad c(t) = 0.2(|[t/100] - 5| + |5 - ([t/20] \bmod 10)|)$$

- Variable t specifies actual generation in a range between 1 and 1000.
- $c(t)$ is actual ideal solution
- The variable x is represented by a bit-string of length 31 and normalized to give a value in the range between 0 and 2.

The goal is to keep on tracking the value of $c(t)$ and minimize the function $g(x,t)$. For more information please look at [14]. Here is a picture of the Ošmera's function.

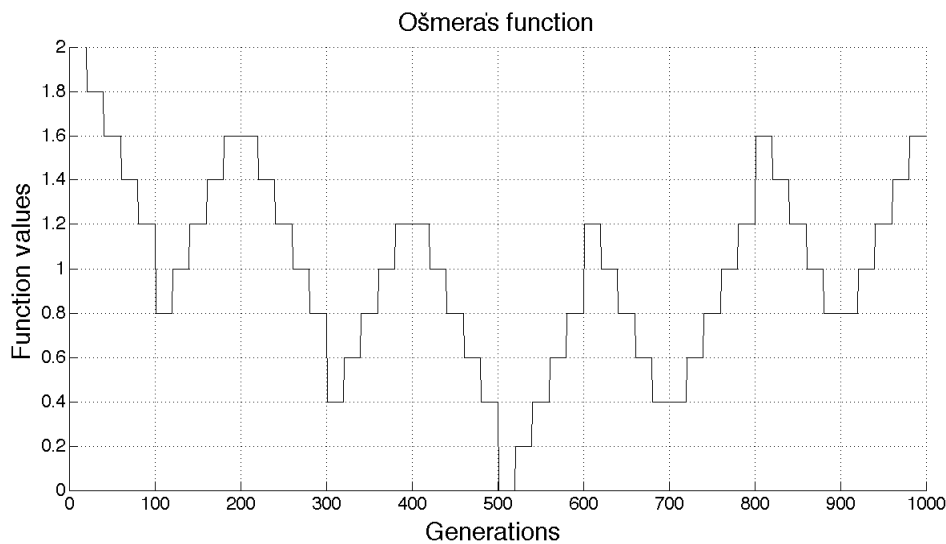


Figure 4.1.e – Ošmera's Function

4.1.3 Deceptive Functions

These functions were developed specially for GA testing purposes, the testing is based on a *schema theorem*, explained for example in [11] and [3].

Input into these functions is a binary string of length 4. For simplicity we can say that the GAs are attracted by the genomes with high fitness. But these genomes are not similar to the ideal solution. Opositely, similar solutions to the ideal one have very low fitness, so these problems are significantly difficult for GAs. More information in [5].

DF3 - Deceptive Function 3

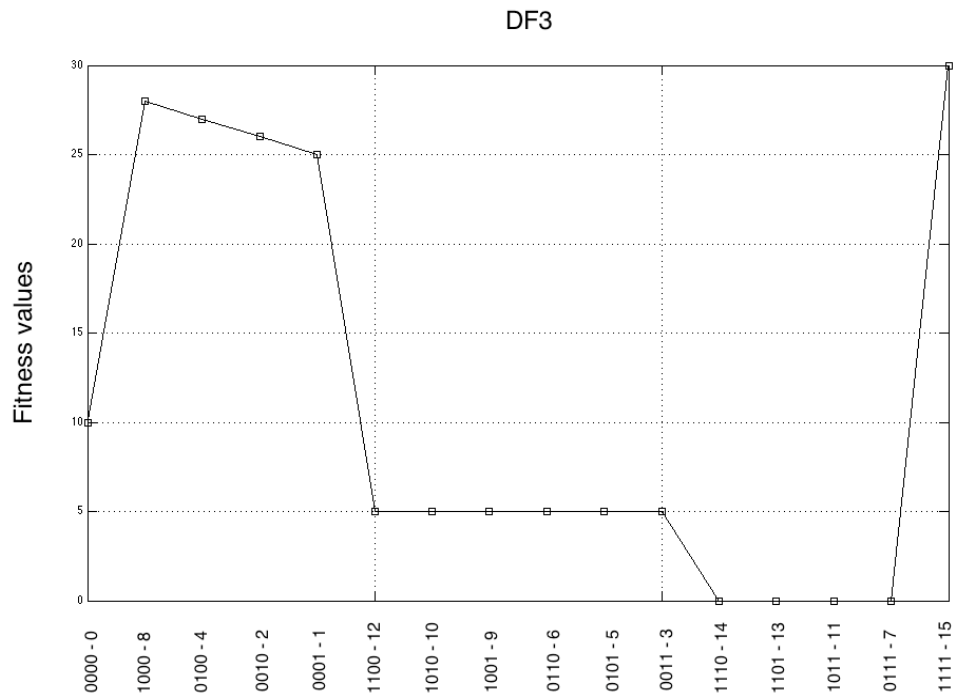


Figure 4.1.f – Deceptive Function 3

PDF – Partially Deceptive Function

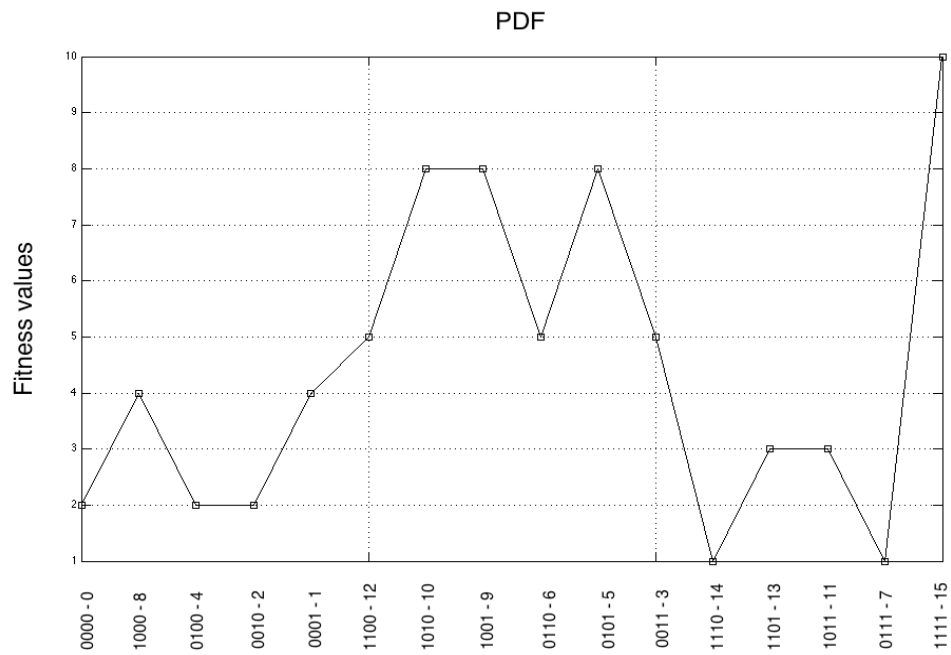


Figure 4.1.g – Partially Deceptive Function

4.1.4 Other Binary Problems

Royal Road

In the Royal Road problem there is the ideal solution composed from particular blocks of defined length. Each block contributes the number equal to its length to the fitness value only if all the genes have value of 1. Here is the example with 10 blocks of length 16. So the ideal genome has 160 bits and all of them have value of 1. The fitness of such ideal genome is also 160. For more information use [13].

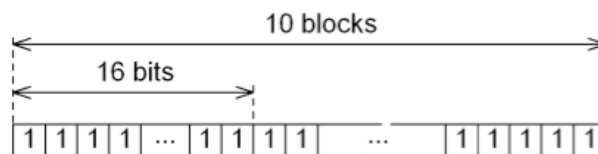


Figure 4.1.h – Royal Road problem

[3]

Hiff

The *Hierarchical If and Only If* problem is composed from a tree of local optimas, sorted from a weak to the very strong ones. Here is the principle of function:

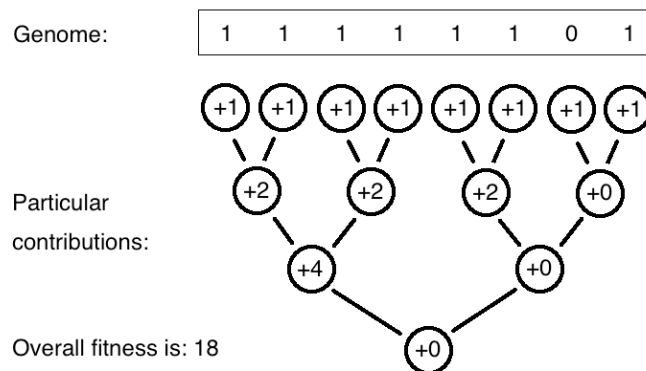


Figure 4.1.i - Hierarchical If and Only If problem

Leaf nodes, corresponding to single genes, contribute to the fitness by 1, each inner node is interpreted as 1 if and only if its children have both the same value, in such cases the inner node contributes to the overall fitness by a 2^n , where n is the actual depth in the graph (i.e. distance from the actual node to its antecedent leaves). Otherwise the value of node is interpreted as zero and its contribution to the fitness is zero. So this problem has two ideal solutions: entire genome is composed from ones, or zeroes. For for more information please look at [12] or [14].

4.2. Basic Configuration of CGA

It is a well known fact, that the configuration of any EA is a bit difficult and doubtful task and this case is not any exception, so the major parameters of this topology will be set up experimentally at first. The other, less important, parameters were set up defaultly before the experiments. In the following text I will consider these parameters as a default, if I won't specify something different.

Table 4.2.a – Basic Configuration of CGA topology

The parameter	value	units
Number of islands	4	islands
Migration interval	20	generations
Number of migrants	25 %	population
Number of migrants A -> B	100 %	population
Number of evaluations	200'000	evaluations
Number of generations	1500	generations
Island model	synch.	synch./asyn.
Wait before first migration	20	generations
Selection type	best/worst	outgoing/in

The CGA topology is configured to quit when it reaches the maximum number of generations, or the maximum number of fitness evaluations. The generation restriction was set big enough, so we can compare the topologies with the different number of islands correctly. The *selection type* row specifies that the outgoing individuals are those best, and individuals to be replaced are the worst ones.

Table 4.2.b – Basic configuration of the problem

The parameter	value	units
The individual	bit-vector	
Genome length	128	bits
Problem	DF3	
Fitness	simple	doubles
Normalize fitness	no	
Stop when the ideal individual found	no	

Table 4.2.c – Basic configuration of common island

The parameter	value	units
Population size	50	individuals
Elitism	1	individuals
Crossover type	2	points
Crossover probability	80	%
Mutation probability	100/genome length	%
Selection type	Tournament	
Selection size	2	Individuals
Seed	random	
Send the number of evaluations	yes	
Send the num. of evals. modulo	1	generations

Table 4.2.d – Basic Configuration of the island A

The parameter	value	units
Population size	50	individuals
Elitism	0	individuals
Crossover type	1	points
Crossover probability	0.0	%
Mutation probability	50	%
Selection type	Best selection	
Seed	random	
Send the number of evaluations	yes	
Send the num. of evals. modulo	1	generations
Source switching modulo	20	generations
Wait before the first soure switching	0	generations

The island topology implemented in the ECJ library is unable to count the number of fitness evaluations. One of my modifications adds to the ECJ ability to stop the evolutionary run after exceeding the specified maximum number of fitness evaluations. This value is counted across the entire island topology.

The inter island communication works in the following way: each of the client islands sends number of evaluated individuals to the server, which summarizes these values and checks whether the maximum of evaluations haven't been exceeded. If yes, the server tells to all of the islands to stop computing and shut down. The parameter *Send the num. of evals. modulo* specifies how often to send these information. Note, that too frequent sending of this information is a big load for the server's communication, and the entire (especially synchronous) evolutionary run is getting slower. For more information look at the *attachment*. Note: The 25% of the population is meant to be 12 individuals in this case.

4.3. Experimental Setting up of CGA main parameters

So I am going to set up the main parameters of the cascade GA topology. Those interested parameters are now: number of islands, number of migrants and a migration interval. I will perform the tests for various configuration of each particular parameter and decide which of them is the best choice. After that I will configure the entire CGA using these three found parameters. The problem could be that each of them works well separately, but their combination can behave worse than each of them. Testing of all the possible combinations would be very difficult. So I will choose some acceptable values of these parameters and each of them I will test on 10 runs of CGA. These tests will be performed on the DF3 problem, which is difficult enough to give some interesting results.

Note: when considering the default genome length and the length of the DF3 function, the maximum value of the fitness function is: $128/4*30 = 960$.

Note: Each of the following tables holds the best fitness values reached on given islands. Unless specified otherwise, all of these values are obtained from ten evolutionary runs.

4.3.1 *Number of Islands*

There were these numbers of islands chosen: 4, 6, and 8.

CGA with 4 Islands

Table 4.3.a – Number of Islands - 4

	Island A	Island B	Island C	Island D
Best fitness	636	862.4	922	926.7

approximate number of generations: 1000

For the illustration of the CGA topology functionality here are the graphs of average and the best fitness values on the particular islands. Note: these graphs comes from one typical run of CGA, the values are not average as usual. For better readability, I have cropped the X-axes to the maximum number of 500 generations.

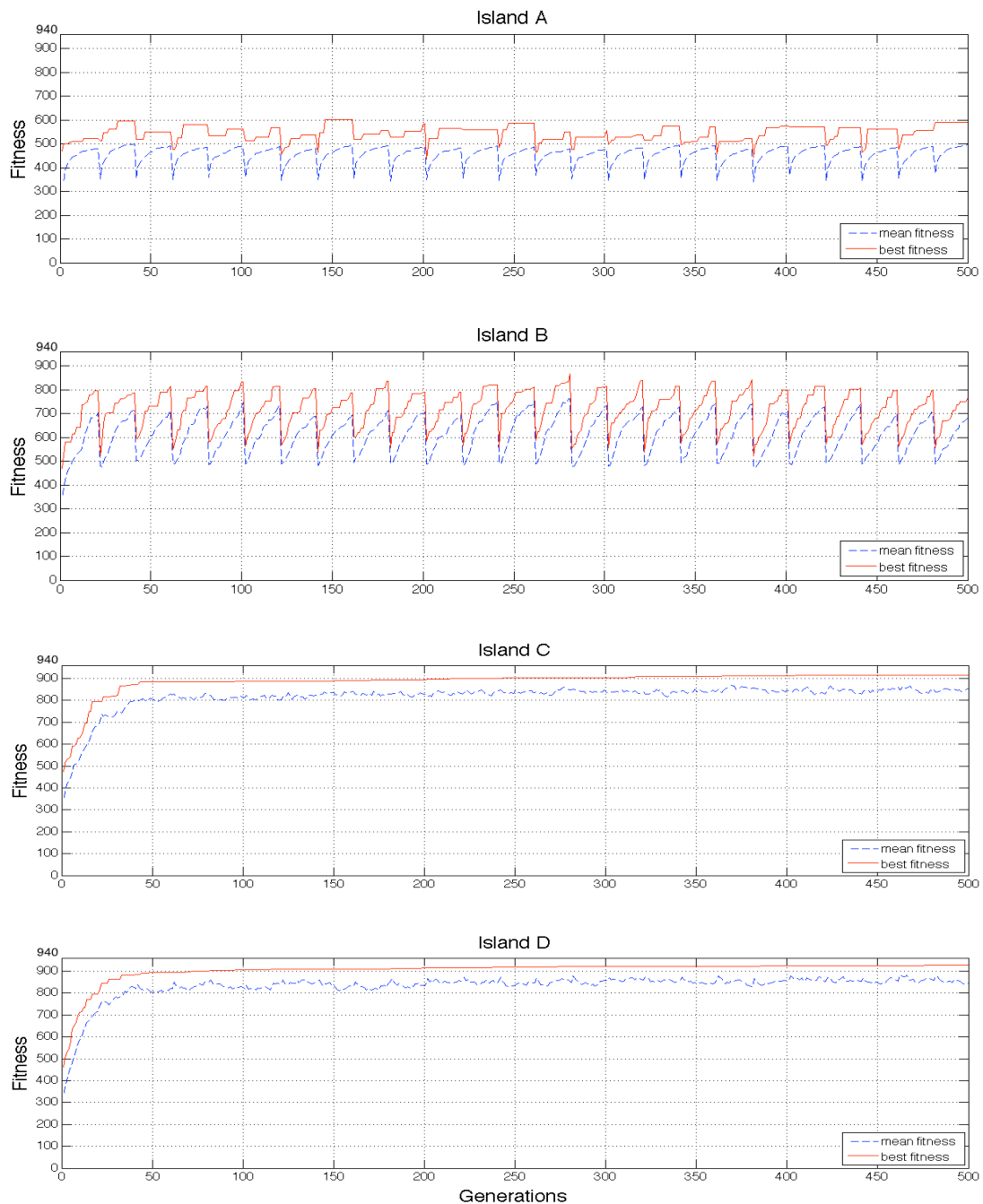


Figure 4.3.a – DF3 task solved by 4-island CGA (typical run)

On the island A there is switching between breeding sources visible, it means between random, and pseudo random generating of individuals. From the graph of island B we can see the migration interval. The islands C and D are similar to each other.

CGA with 6 Islands

Table 4.3.b – Number of Islands - 6

	Island A	Island B	Island C	Island D	Island E	Island F
Best Fitness	623.8	858.6	913.1	921.2	928.5	931.8

approximate number of generations: 668

CGA with 8 Islands

Table 4.3.c – Number of islands - 8

	A	B	C	D	E	F	G	H
Best fit.	619.2	859.3	913.8	922.9	925.8	930.7	935.1	939.4

approximate number of generations: 502

CGA with 10 Islands

Because of the best results given by the 8 island topology, I was interested in the behaviour of 10 island topology.

Table 4.3.d – Number of Islands - 10

	A	B	C	D	E	F	G	H	I	J
Best:	614.1	856.6	902.7	915.7	922.7	927.4	930.3	935	939.5	940.7

approximate number of generations: 402

From these results we can see that the best number of islands for the DF3 function is the last one, 10. But the difference between result given by 8 and 10 island topology is very small and I think, that the other properties of the topology with 10 islands will be significantly worse, for example for solving dynamic problems it could be absolutely useless. So for the final configuration I will use **just 8 islands**.

4.3.2 *Number of Migrants*

The tested numbers of migrants are: 5, 10, 25, 40 and 45 individuals, where the population size is constantly set to 50 individuals.

I will test these parameters on the default CGA topology.

Note: these numbers do not specify the number of migrants from the island A, it is always 100% of the population. Average number of generation is always 1000 here.

5 migrants

Table 4.3.e – Number of Migrants - 5

	Island A	Island B	Island C	Island D
Best fitness	626	862.7	922.6	929.7

10 migrants

Table 4.3.f – Number of Migrants - 10

	Island A	Island B	Island C	Island D
Best fitness	627.3	862.8	915.1	924.8

12 Individuals

Table 4.3.g – Number of Migrants - 12

	Island A	Island B	Island C	Island D
Best fitness	636	862.4	922	926.7

25 migrants

Table 4.3.h – Number of Migrants - 25

	Island A	Island B	Island C	Island D
Best fitness	632.2	869.7	928.1	933.9

40 migrants

Table 4.3.i – Number of Migrants - 40

	Island A	Island B	Island C	Island D
Best fitness	625.5	870.9	935.5	940.7

45 migrants

Table 4.3.j – Number of Migrants - 45

	Island A	Island B	Island C	Island D
Best fitness	621.8	864.2	938.4	942.8

We can see that the migration of a **90% of the population** is here the best solution, so I will include this parameter in the future tests.

4.3.3 Migration Interval

Finally, we need to find out the ideal migration interval, this parameter is depended on the number of migrants, so the only difference against the default configuration is the result from the previous experiment: **number of migrants = 45**.

The analyzed migration intervals are now: 10, 20 and 50 generations.

10 generations

Table 4.3.k – Migration Interval - 10

	Island A	Island B	Island C	Island D
Best fitness	644.3	807.2	928.9	936.3

20 generations

Table 4.3.l – Migration Interval - 20

	Island A	Island B	Island C	Island D
Best fitness	621.8	864.2	938.4	942.8

50 generations

Table 4.3.m – Migration Interval - 50

	Island A	Island B	Island C	Island D
Best fitness	624.6	899	934.7	938.4

The result is that when we are migrating too frequently, the islands don't have enough time to evolve some good individuals. When using too big migration interval, there is not enough divergent population in the topology.

So the new **migration interval = 20 Generations**.

4.4. Configured CGA vs. PGA

4.4.1 CGA configuration

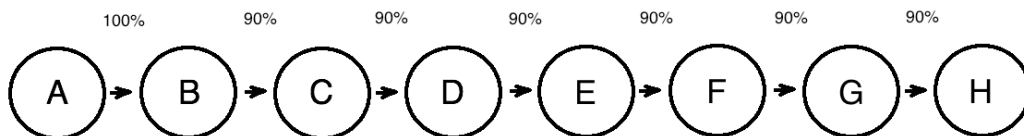
So I have configured the CGA topology using these new values obtained from the previous chapter. For sure I present here the new configuration table.

Table 4.4.a – The Configured CGA topology

The parameter	value	units
Number of islands	8	islands
Migration interval	20	generations
Number of migrants	90 %	population
Number of migrants A -> B	100 %	population
Number of evaluations	200'000	evaluations
Number of generations	1500	generations
Island model	synch.	synch./asyn.
Wait before first migration	20	generations
Selection type	best/worst	outgoing/in

The other parameters of the CGA are unchanged. Now this configuration can be considered as the final one, so I will start with testing on some of the other tasks for the best review of all the CGAs properties.

Number of Migrants:



Migration Interval: 20 Generations

Figure 4.4.a – CGA configuration

4.4.2 *PGA Configuration*

For the comparison purposes I have configured another similar island GA. This PGA will be configured as a common parallel GAs used to be, but I tried to keep as many parameters the same as in the configured CGA as possible. The PGA uses one directional ring with the same number of islands as in the CGA.

Table 4.4.b – Configuration of PGA topology

The parameter	value	units
Number of islands	8	islands
Migration interval	20	generations
Number of migrants	90%	population

The other parameters are similar to the default configuration of the common island in the CGA topology.

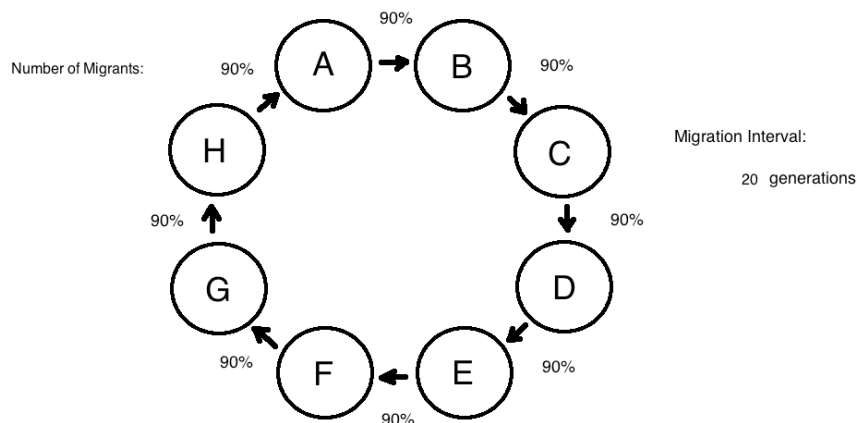


Figure 4.4.b – PGA configuration

4.4.3 *Tests*

Here are the tested tasks: DF3, HIFF, Ošmera's function, Rastrigin's function, Schwefel's function. In each of the tests there is the table with two rows, first row describes the results given by CGA, and the second one represents the PGA topology. The values are again the fitness value of the best individual found on the particular island. These values are average over the 10 evolutionary runs.

The configuration of Schwefel's function is in the table. The configurations of other real functions were similar with respect to the recommended range for input variables.

Table 4.4.c – Real Function Configuration - Schwefel

The parameter	value	units
The individual	bit-vector	
Genome length	60	bits
Problem	Schwefel	
Lower Bound	-500	double
Upper Bound	500	double
Number of input variables	2	
Tolerance for the ideal solution	1e-9	fitness

This mean that each of the genes is divided in two parts, so each of the input variables is coded using 30 bits into the specified range. So that the GA cannot generate forbidden solutions, i.e. numbers out of range.

The HIFF problem has the genome length by default 128 bits.

Configuration of the both the GAs is a bit changed when solving the **Ošmera's function**. This problem is dynamic with change of the optima every 20 generations, so the **migration interval** is set to the lower value of **5** generations.

The other parameters are the same as the default ones. The average nuber of generations is always 490 here.

DF3

Table 4.4.d – CGA vs. PGA - DF3

	A	B	C	D	E	F	G	H
CGA	618.5	851.3	927.1	932.8	935.3	788.8	941.7	944.3
PGA	946.8	946.4	946.2	946.7	946.6	946.6	946.8	946.6

Ideal fitness: 960

HIFF

Table 4.4.e – CGA vs. PGA - HIFF

	A	B	C	D	E	F	G	H
CGA	285.8	444.2	721.8	733.6	757.6	768.8	786.4	796
PGA	814.4	814.4	814.4	814.4	814.4	814.4	814.4	814.4

Ideal fitness: 1024

Schwefel's function

Table 4.4.f – CGA vs. PGA – Schwefel's function

	A	B	C	D	E	F	G	H
CGA	-837.110017	-837.965566	-837.965769	-837.96576	-837.965765	-837.965769	-837.965769	-837.965769
PGA	-837.965758	-837.965809	-837.948019	-837.94800	-837.965805	-837.965725	-837.963980	-837.958985

Ideal fitness: -837.9658

Ošmera's function

In this case the comparison of best fitness is useless, so I will bring out the graphs for each of the islands.

On the first set of graphs there will be the Ošmera's function compared to the best individual found in the actual generation. These values are taken from one typical evolutionary run. On the Y-axis there are the values of the Ošmera's function, and the X-axis represents the number of generations.

The second set of graphs will contain series with an absolute distance of the best solution found on a particular island from the actual value of Ošmera's function. These values will be average from 10 evolutionary runs.

Table 4.4.g – Additional Configuration table for the Ošmera's problem

The parameter	value	units
Migration interval	5	generations

CGA - The Ošmera's function compared to the best solution

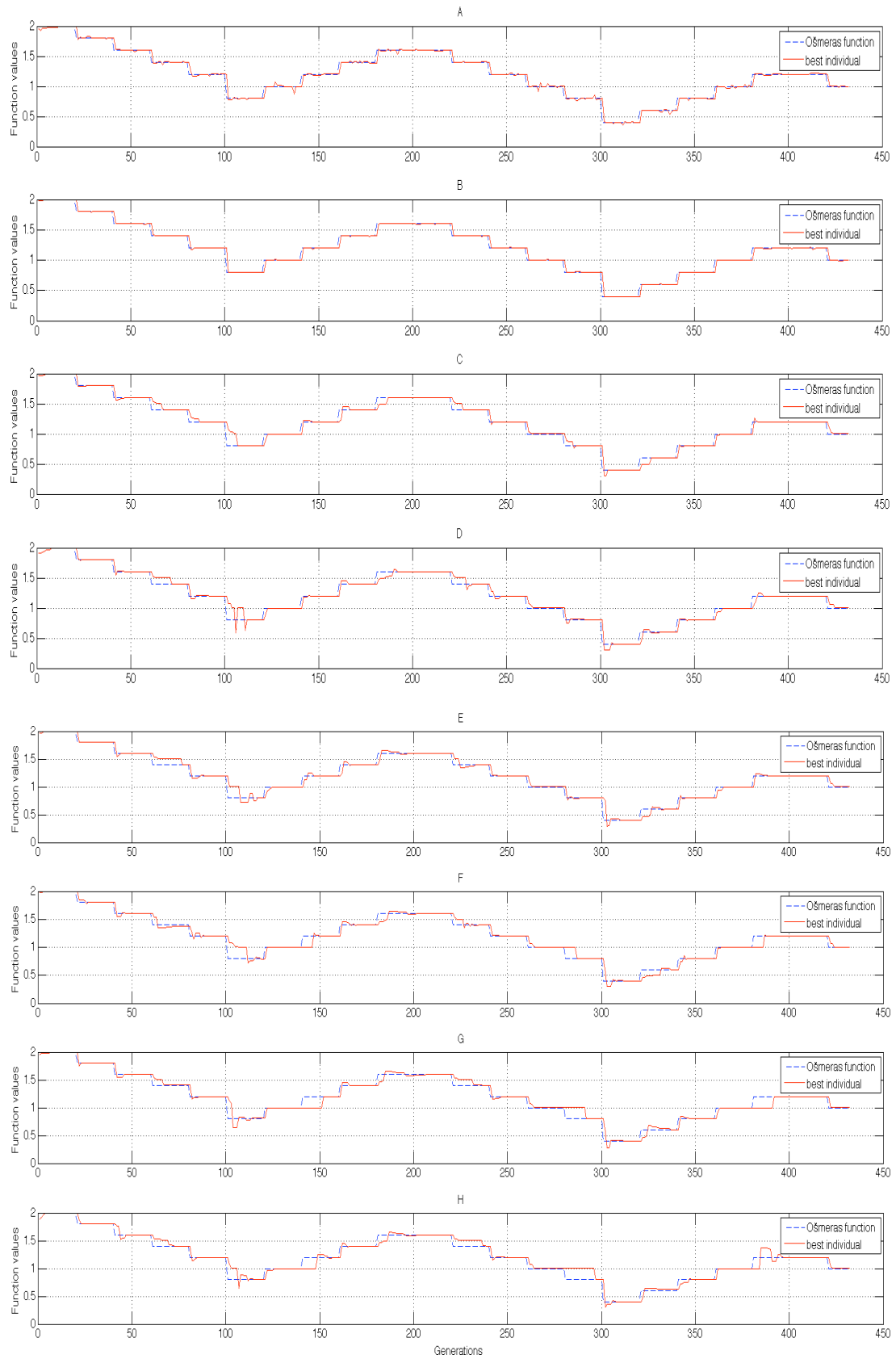


Figure 4.4.c – Ošmera's function solved by the CGA (typical run)

PGA - The Ošmera's function compared to the best solution

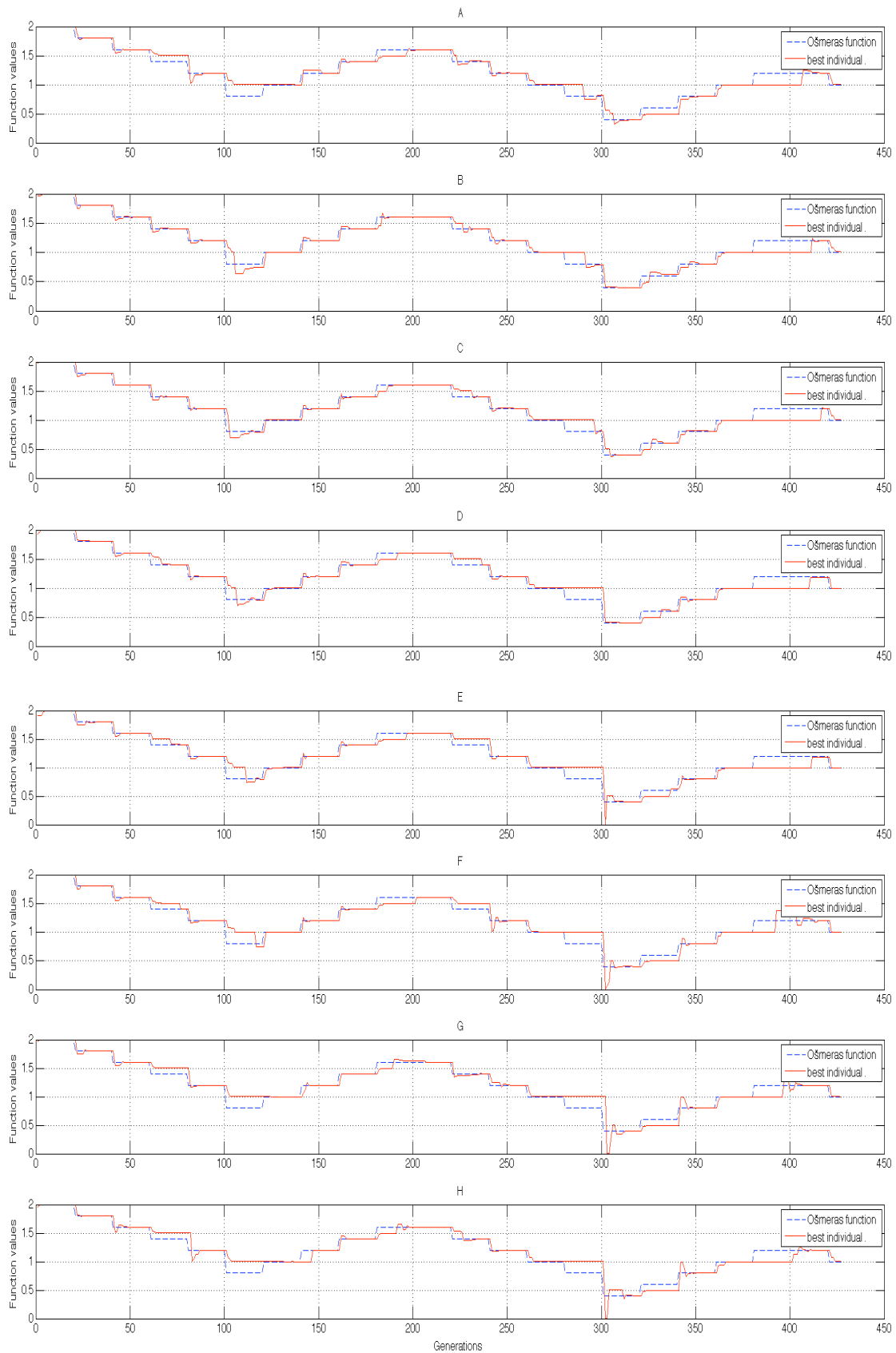


Figure 4.4.d – Ošmera's Function solved by the PGA (typical run)

CGA - The absolute error of the best solution found

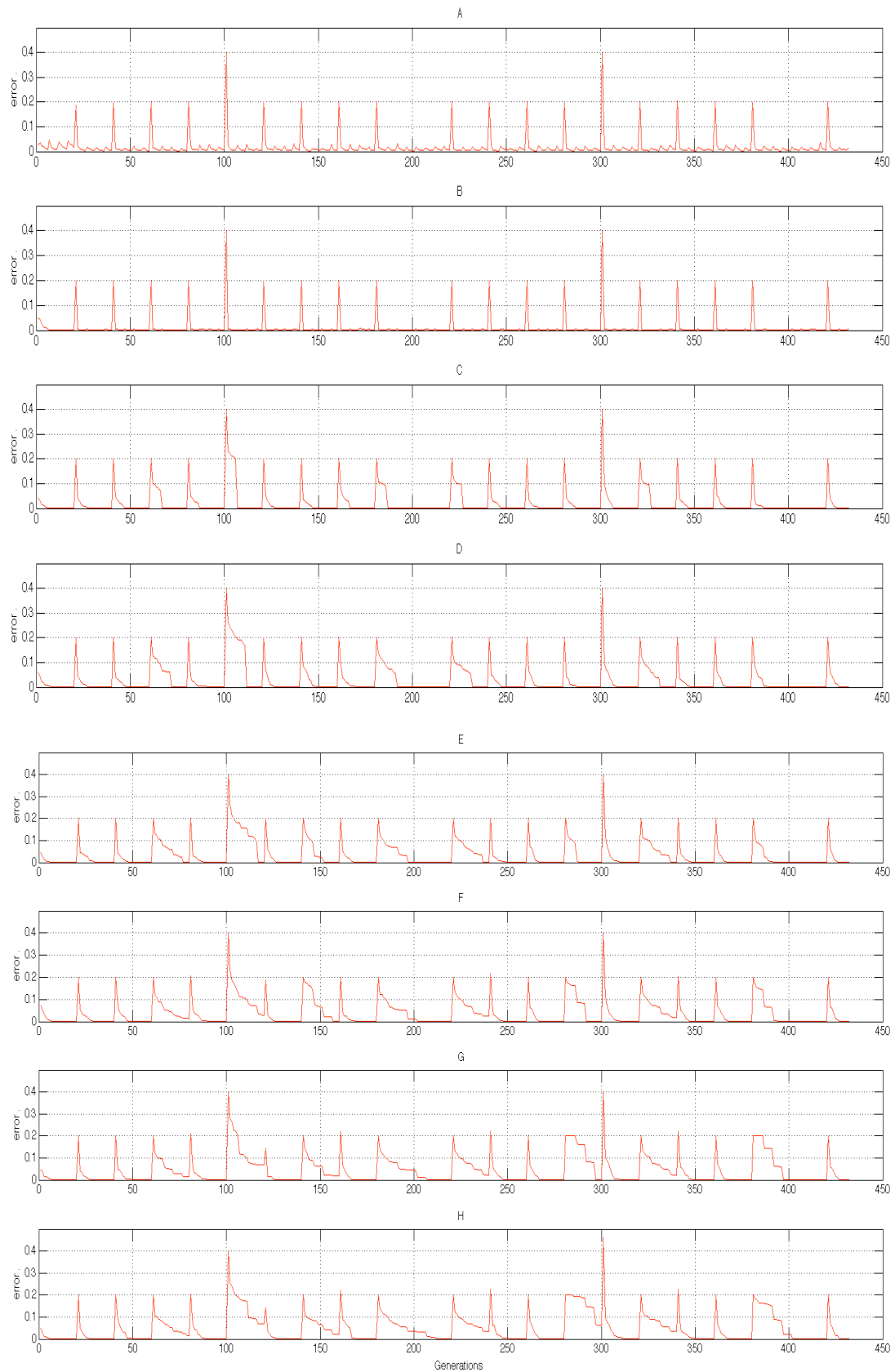


Figure 4.4.e - absolute error from ideal solution on CGA topology (average values)

PGA - The absolute error of the best solution found

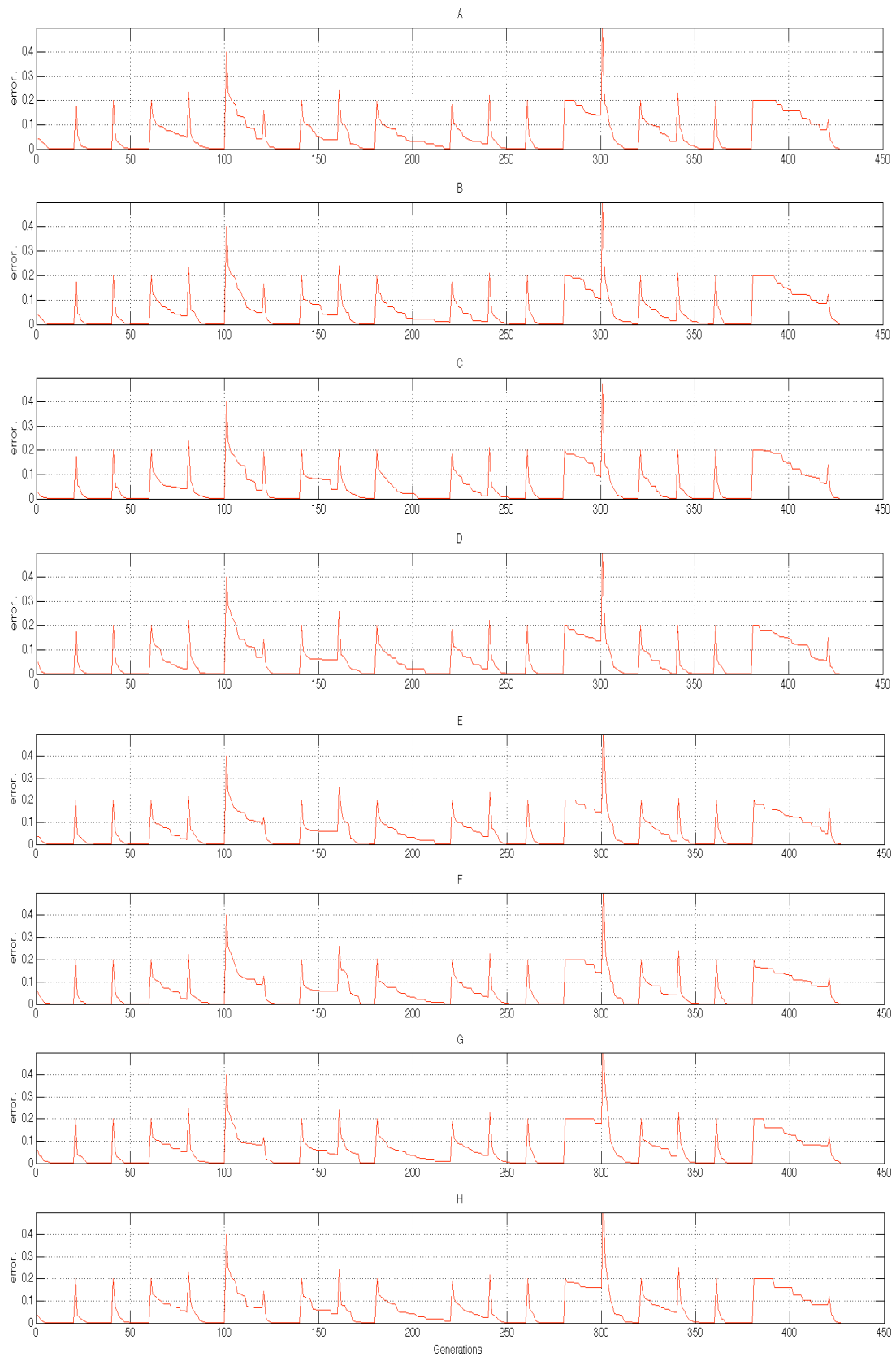


Figure 4.4.f- absolute error from ideal solution on PGA topology (average values)

4.5. Smaller CGA topology vs. PGA

Because of not very good results given by the CGA topology with 8 islands, it was decided to test some smaller CGA topology with bigger migration, here is the configuration table:

4.5.1 *Smaller CGA Configuration*

Table 4.5.a – Smaller CGA Configuration

The parameter	value	units
Number of islands	4	islands
Migration interval	20	generations
Number of migrants A -> B	98 %	population
Number of migrants B -> C	98 %	population
Number of migrants C -> D	25 %	population

4.5.2 *Smaller PGA Configuration*

Table 4.5.b - Smaller PGA configuration

The parameter	value	units
Number of islands	4	islands
Migration interval	20	generations
Number of migrants A -> B	90 %	population
Number of migrants B -> C	90 %	population
Number of migrants C -> D	90 %	population
number of migrants D -> A	90 %	population

DF3

Table 4.5.c - DF3 task solved by Smaller Topologies

	Island A	Island B	Island C	Island D
CGA	625.8	918.3	927	935.4
PGA	937.6	937.6	937.8	937.8

HIFF

Table 4.5.d - HIFF task solved by Smaller Topologies

	Island A	Island B	Island C	Island D
CGA	293.2	609	636.8	983.2
PGA	717.2	717.2	717.2	717

4.6. CGA topology for Dynamic Problems

The CGA topology proved to very useful for solving dynamic problems. But both of the previous topologies wee too long, so I have assembled only two-island CGA topology, and I will test its behaviour on the Ošmera's problem. The results will be compared with the similar PGA topology.

4.6.1 *Small CGA Topology*

Table 4.6.a – Small CGA for Dynamic Problems conf.

The parameter	value	units
Number of islands	2	islands
Migration interval	5	generations
Number of migrants A -> B	95 %	population

4.6.2 Small PGA Topology

Table 4.6.b – Small PGA Topology for Dynamic Problems conf.

The parameter	value	units
Number of islands	2	islands
Migration interval	5	generations
Number of migrants A -> B	95 %	population
Number of Migrants B -> A	95 %	population

Here are the resulting graphs from the tests made on the Ošmera's function. These courses are typical ones because of distortion caused by potential averaging.

CGA

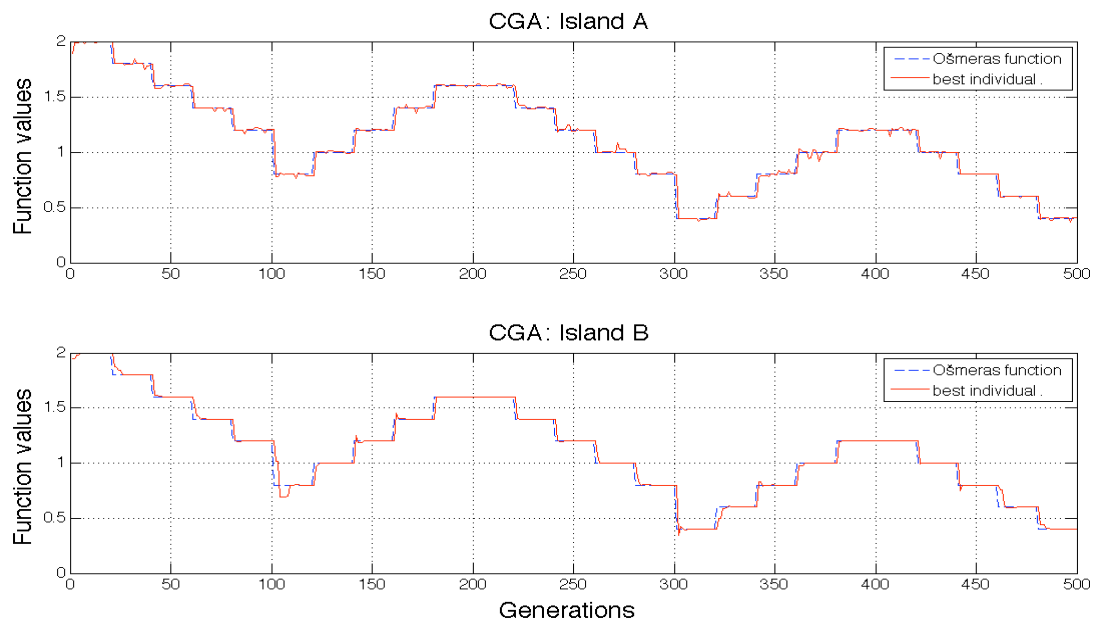


Figure 4.6.a - Ošmera's Function Solved by the Small CGA Topology (typical run)

PGA

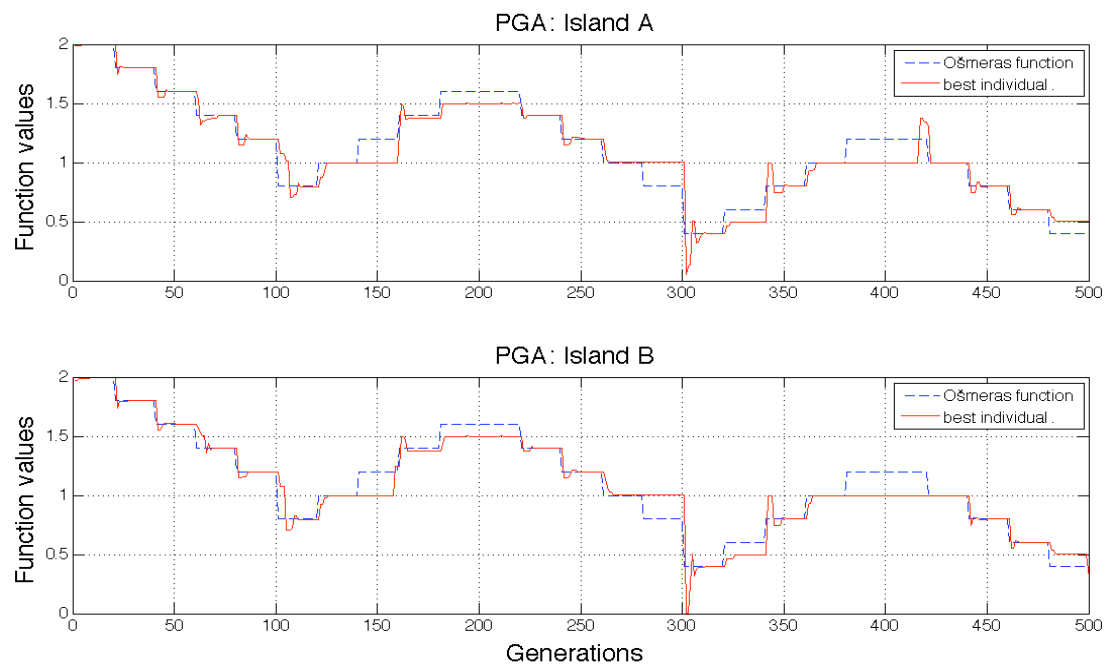


Figure 4.6.b – Ošmera's Function Solved by the Small PGA Topology (typical run)

For better readability here are graphs of absolute error. This error is meant to be the absolute distance of the best solution from the actual value of Ošmera's function. These values are average across 10 evolutionary runs.

CGA

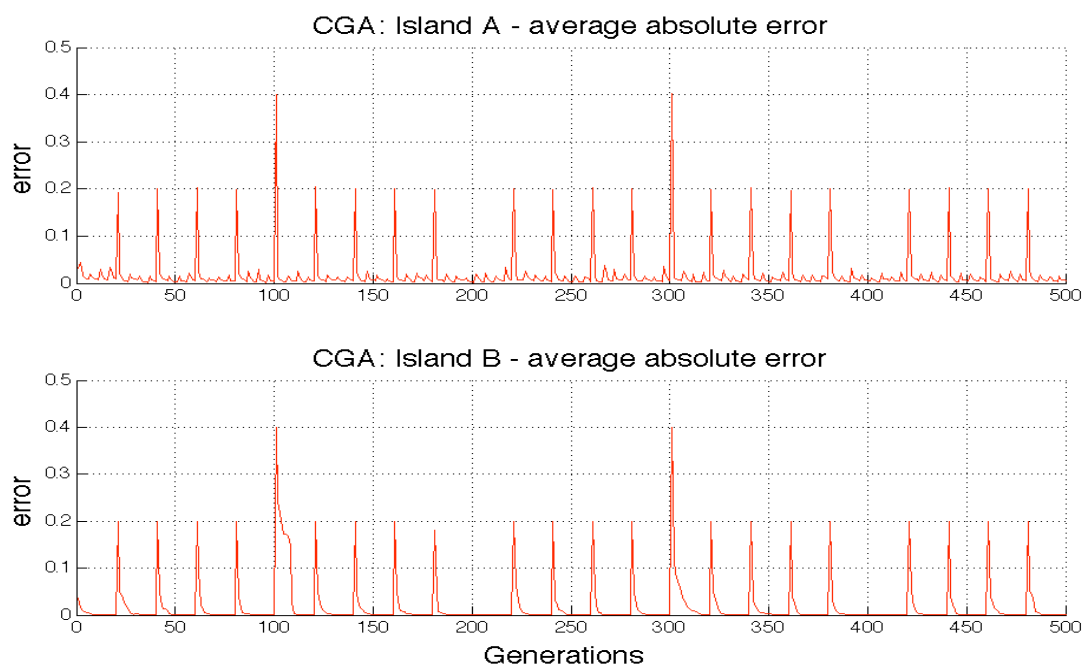


Figure 4.6.c - CGA - absolute error of the best solution found (average values)

PGA

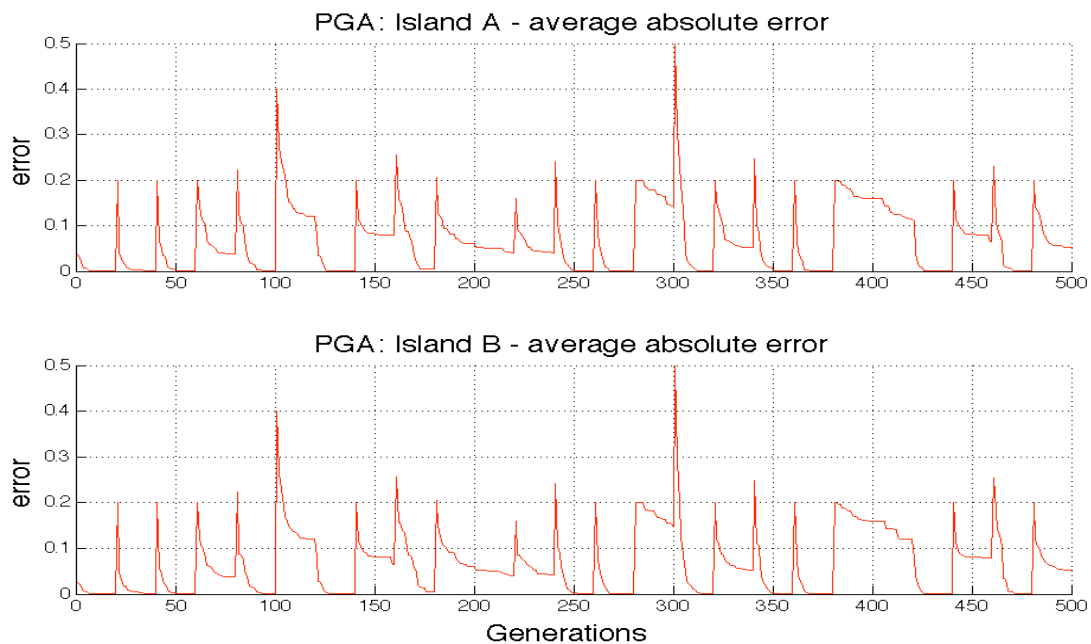


Figure 4.6.d - PGA - absolute error of best solution found (average values)

4.7. Discussion

Configuration of CGA

In the beginning I tried to configure the CGA topology by testing the influence of particular parameters on behaviour of whole algorithm. This is simplified, but also inexact, approach. I found out that the longer cascade topologies shows better results despite the smaller number of generations. I verified that the number of migrants and the migration frequency should be big enough to sustain the population divergency and for a good propagation of newly generated individuals through the entire topology. In the other hand when the migration is too big, the immigrants decay the average fitness in the topology.

Experiments

The experiments performed on such configured CGA showed that the last island in the topology was qualified as the output from the topology correctly. In this island there can be the best individuals found. And now I will try to judge the assumption that this topology holds its populaitons divergent enough. The experiment results showed that the CGA has similar qualities as the common PGA, but the output is usually a little worse. On the DF3 problem, there are almost identical results, but on the HIFF problem there are bigger differences, which shows that the common approach is better decision in this case. Theese two algorithms show the similar results when solving the dynamic problem. In order to reach some better results I configured a smaller CGA topology including only four islands with huge migration

and tried to compare with the similar PGA. Despite the use of too big migration, which downgraded the PGA noticeably, the results on the DF3 problem were similar. The HIFF function showed here that the PGA was not configured very well. The experiments also showed that when solving real functions of two variables, both of the algorithms achieve the satisfactory results after several generations.

So I think, that the divergence in the populations in the CGA is bigger than on some common PGAs, but with the combination of fitness decay caused by the new immigrants, the results are almost comparable. I assume that this fact may be caused by the inexactly set parameters of CGA, but solving this problem would require much more experiments with the configuration.

During the course of the experiments I noticed that some short CGA topology could be very useful for solving dynamic experiments. So I composed a small CGA topology including only two islands and tested it on the Ošmera's dynamic problem. This small two-island configuration proved very good behaviour compared to the common similar PGA topology. The island A supplied the target island B with the sufficient number of divergent individuals, so the topology had very short reaction time and tracked the ideal solution with high precision.

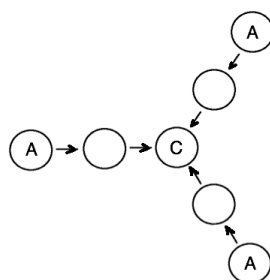
4.8. Conclusion

The aim of this thesis was to propose and review the Cascade Genetic Algorithm, but the most difficult part lied in use of very sophisticated Java library for evolutionary computation - ECJ, understanding and improving them for this specific purpose.

Testing of CGA demonstrated that this approach is not so good as I supposed. The commonly used parallel topologies shows slightly better results, but I think this doesn't mean that the idea of cascade topology is wrong, but something small is just missing.

This small thing could be for instance some modification of selection method, able of selecting one parent from new immigrants and one parent from original residents of given island. This feature could provide better population diversity by improving the combination of old and new solutions.

Another solution could be in use of some different topology based on the cascade principle, for instance something like this:



Where an islands A would have the similar purpose, as the island A in the CGA and an island C could be the output. In the island C there could be mixed divergent individuals coming from several independent sources at the same time.

But a bright future for some kind of CGA I see in solving dynamic problems, where this approach provides considerably better results than the common PGAs.

Attachments

This chapter is a brief description of the things I've implemented in Java and things I have modified or added in the common version of ECJ. For example in order to setting up new functions, the custom ECJ obtained many parameters used for configuring the evolutionary run. I will mention them and their meanings here also. I will also try to describe some of the small bugs in the original ECJ project. At the end I will describe where the experiments mentioned in this thesis (and many other) are stored. Finally, I will describe the simple way to launch these experiments.

Attachment A: List of ECJ modifications

Here is the short list of upgrades I've made on the custom ECJ library. It is sorted by the importance of classes in the library. This list could be useless for many purposes, but if somebody would like to continue with this work, I hope this could be helpful.

A.1. Upgrades

IslandExchange:

- repaired error in inter-island synchronous communication:
 - o some send individuals were not accepted by the target mailbox occasionally, so that the island didn't receive any immigrants. These lost individuals were used to appear in the mailbox in some random further generation
 - o So the target island sometimes received nothing, and sometimes received two groups of immigrants despite its synchronous communication.
 - o Serious problem is receiving two groups of immigrants when migrating one half or more population, because population on target island is going to be deleted completely
 - o Solution is that we can force the islands to wait for some new immigrants (i.e. at least one) by parameter *force-receive* (which is set true by default)
 - o note.: this parameter must be set to *false* in the island A in the CGA topology
- added check if number of incoming individuals is not bigger than population of island
- unsolved selection methods – Island can possibly become irresponsible when selecting too big part of population
 - o in myBestSelection it is solved correctly
- added ability of selecting worst N individuals to be replaced by new incoming immigrants (EvaluateII)
- adding statistics about number of fitness evaluations across all Islands in given topology

- added stopping criterion for number of fitness evaluations across the all of islands in the topology (handled by th server thread)
- added variables:
 - *sendEvalsModulo, sendEvals, P_SEND, P_SEND_MODULO, EVALUATED, force-recieve, P_FR, forceRecieve*

IslandExchangeServer:

- added ability to count fitness evaluations remotely reciaived from islands
- added ability to stop entire evolutionary run when *maxNumOfEvals* exceeded
- every island counts number of fitness evaluations, and sends it to the server, value is temporary and zeroed by method *preBreedingExchangePopulation* in *IslandExchange* after every send
- *myFitness.numOfEvals++* is called in particular implementations of problem
- partially solved problem with slow synchronization when counting fitness evaluations: when the server founds at least one message containing number of evaluations, switches itself to a faster mode, that means that the sleeping time between messages accepting and processing is cuted down to one millisecond (stored in a variable *FASTER_SERVER*)
- added ability to define whether the server should print out its messages to the command line (these are starting: „Server: „)
- added variables:
 - *P_MAX_NUM_OF_EVLS, GOODBYEII, alreadyReadGoodByeII, EVALUATED, maxNumOfEvals, numOfEvals, FASTER_SERVER, mySleep, printServer*

IslandExchangeMailbox:

- Improved catch of *IO Exception*
 - This exception was written on the end of ER, when someone tried to shutdown the mailbox, and mailbox was trying to comunicate with an source island.
 - Method *mailbox.shutdown()* is called on the end of evolutionary run typically, and at this time are other islands closed usually. So when the exception happens, we will take a look whether the evolotuinary run is on the end, and if yes, we just return without any warnings.

EvolutionState:

- added subclass *myFitnessInfo*
- added public instance of *myFitnessInfo* named ***myFitness*** containing:
 - *countEvals* – tells to the server whether to count evaluations
 - *numOfEvals* - actual number of unsend evaluations (island)
 - *useMyFitness* - whether we are using *myFitness* (realFunctions)
 - *min,max* - maximum range for fitness values (printing)
 - *bordersDefined*-whether these ranges were set
 - *maximise* - whether it *myFitness* should be maximised, or minimised (unused by now, for highlighting ideal fitness in graphs)

- dynamic - whether the solved problem is dynamic
- MyName - the name of each island (graph and stats)
- these values are used mainly by these classes:
 - IslandExchanger
 - myMatlabStatistics
 - printBest
 - RealFunction, and all implementations of problems (sets min,max..)
 - dynamic problems – we cannot copy unevaluated elites here! It is handled in *SimpleBreeder* in method *loadElites*, if the problem is dynamic, elites are copied to new population, but they are marked as unevaluated.
- Added ability to shut down the entire evolutionary run from „maximum number of evaluations exceeded“, independently on „found ideal individual“ reason. Added variables are:
 - *quitOnRunCompleteII* - specifying whether to quit if maximum of evaluations across the entire topology was exceeded
 - *P_QUITONRUNCOMPLETEII* – string specifying necessary parameter
- So the *quitOnRunComplete* and the *quitOnRunCompleteII* are completely independent on each other.

SimpleEvolutionState:

- added second use of method *evaluatePopulation* (EvaluateII)
 - EvaluateI is not surrounded by pre and *postBreedingExchange* statistics, so it is not as exact as it should be
- added variable:
 - *forFirstFound* – indicates whether the ideal solution was found for the first time, if yes and *quitOnRunComplete=false* do not print it again

Individual:

- adding public variable *myFitness* for saving original, untransformed value of fitness for correct plotting (because of problems with using negative-values of fitness)
 - o used mainly in class *RealFunction* (where we need minimisation of function with negative values, so the transformation was necessary)

VectorSpecies:

- added to parameter *mutation-prob* ability to recognize string „*1/genome-length*“, which automatically determines the value of mutation probability
- if this string used, *VectorSpecies* prints it out to standard output

BitVectorIndividual:

- in method *defaultMutate(..)* added parameter *Probability*, giving the ability to specify custom mutation probability. (Probability, that each gene in entire genome will be bit-flipped.)

SimpleStatistics:

- printing the name of each island added to the command line

A.2. New Classes

myBestselection:

- selection method, that can select exactly N best / worst individuals from population
- solved few problems with possibly island freezing when selecting main part of population

myFitnessInfo:

- some additional data to *EvolutionState*, holds information about:
 - number of evaluated individuals (between sending to server)
 - whether to use *myFitness* (original fitness function)
 - bounds for fitness values (*Fitness* or *myFitness*)
 - if these bounds were defined
 - whether we want to maximise *myFitness*
 - whether we want to count number of evaluations across all of the Islands (whether to send numbers about evaluations is locally stored in every particular island as a *sendEvals*) if this is set to false, server counts (eventually incoming) number of evaluations, but don't inform islands about it
- in this application it is public variable accessible by: *state.myFitness...*

A.3. New Breeding Pipelines

myVectorMutationPipeline:

- has almost the same behavior as (vector/breed/) *VectorMutationPipeline*
- added parameter: **mutation-prob**, specifying its own mutation probability

mySourceSwitcher:

- has similar function as (breed/) *GenerationSwitchPipeline*
- has two sources, by default it simply connects source 0 with output
- it waits **offset** generations and after that (with period given by **switch-modulo**) switches to source 1, after one generatio it switches back to the source 0

SmallEvaluator:

- it is little hack, because it in fact substitutes the classic SimpleEvaluator by evaluating newly bred individuals (because we need them here evaluated)
- on input it has some individuals, and it just copies them to output and if they are not evaluated, evaluate them
- it is implemented in order to ability of comparison newly bred individuals with some other (the current ones in our case)
- in its own *setup()* method it makes its own pointer to initialized instance of SimpleProblemForm (e.g. Osmera, some child of RealFunction, HIFF, DF3, PDF, etc..) and after that it's able to call required *evaluate()* method

SourceComparator:

- breeding pipeline with one official source, but in fact it has two sources, the second one is current population
- in method *prepareToProduce()* it copies entire current population to its output, and in *evaluate()* method it merges newly incomming individuals with the old population (actually stored on output array - *inds[]*)

myBreedBestSelection:

- it's adapted myBestSelection (by deleting one of overloaded methods *evaluate()*) in order to correct function as BreedingSource
- there was problem with returning value (index pointing to individual in population vs. number of produced individuals)
- note: in *prepareToProduce()* method it starts to remember the number of produced individuals, and increases it every *produce()* function, so this selection method is able to produce consequently entire population

A.4. New Plotting and Exporting Classes

PrintBest:

- prints out an online graph with two fitness series: best Individual from population and average fitness in population

PrintOsmera:

- prints out an online graph with two fitness series: Osmeras function, and the best individual from population

PrintPopulation:

- prints out an online graph with two fitness series: the fitness values of an even population, the fitness values of an odd population, on the x-axis there are numbers of individuals in the population

myMatlabStatistics:

- simple class for exporting data to text file readable for example by matlab
- mainly copied from ecj's *SimpleShortStatistics*
- exports for every subpopulations following variables:
 - generation number
 - if(export-original-fitness=true || useMyFitness =false):
 - mean fitness in actual population
 - best fitness in actual population
 - if(useMyFitness = true)
 - mean *myFitness* in actual population
 - best *myFitness* in actual population
 - if (state.myFitness.iAmServer=true && state.myFitness.countEvals)
 - number of evaluations across the entire population
- note: starting script (usually containing Island D) is configured that after pressing the enter key (on the end of evolutionary run) copies all exported *.stat files to the Matlab's path
- ability to print fitness values to the command line (and to switch on/off them)
- ability to print out the best found individual and its fitness to the command line, after some termination condition found
- ability to print out the osmeras function and the function values of the best individual in each of generations

A.5. Implemented Tasks

my

- PDF
- DF3
- Ošmera's problem
- Knapsack (static/dynamic)
- Royal Road problem

AbstractRealFunction childs:

- Sphere
- Rastrigin
- Rosenbrock
- Schwefel

upgraded:

- Hiff problem

Attachment B: Added parameters

Some new added parameters follow, so if you don't know what is written in the parameter files, you could find the explanation here. For the meaning or use of basic parameters and parameter files please check the ECJ documentation and tutorials [15].

B.1. Exchange

- **num-of-evaluations** (*base = ec.exchange.IslandExchange*)
 - *int*
 - maximum number of fitness evaluations (if not specified, it wont become the stopping criterion, also if `quitOnRunComplete = false`, exceeding of num-of-evaluations will bring up only notification)
 - should be specified in server *.params file
 - placed in **Topology folder**
- **count-evals** (*base = ec.exchange.IslandExchange*)
 - *boolean*
 - tell to server whether to count evaluations or not
 - if false, server still counts incoming numbers from islands, but will not send them back any notification (f.e. about exceeding the maximum number of evaluations)
 - placed in **Topology folder**
 - it is equivalent to `quitOnRunComplete` – if we want to shut down the evolutionary run from number of evals exceeded: this and `quitOnRunComplete` must be equal to true
- **eval-send** (*base = ec.exchange.IslandExchange*)
 - *boolean*
 - parameter for each island specifying whether to send the numbers of fitness evaluations or not
 - placed in **Apps folder**
- **eval-send-modulo** (*base = ec.exchange.IslandExchange*)
 - *int*
 - how often to send information about number of evaluations [in generations]
 - must be specified in particular island *.params files (because it is parameter of `islandExchange` on island, not on server)
 - placed in **Apps folder**
- **select** (*base = ec.exchange.IslandExchange*)
 - *ec.selectionMethod*
 - the selectionMethod to select outgoing individuals from island
 - placed in **Apps folder**
- **select-to-die** (*base = ec.exchange.IslandExchange*)
 - *ec.selectionMethod*

- the selectionMethod used to select individuals to be replaced by new, incoming, from different islands
- placed in **Apps folder**
- ***pick-worst*** (*base = ec.exchange.IslandExchange. select || select-to-die*)
 - *boolean*
 - whether to pick best or worst individuals (by selectionMethod)
 - placed in **Apps folder**
- ***print-server*** (*base = ec.exchange.IslandExchange*)
 - *boolean*
 - whether to print servers messages to the command line
 - messages as: incoming communication from islands, synchronization status, etc..
- ***force-recvie*** (*base = ec.exchange.IslandExchange*)
 - *boolean*
 - this parameter is important only in synchronous inter-island communication and defines whether the island will wait for some individuals in its own mailbox
 - thanks to this is repaired error in the ECJ's communication, and the entire island topology works correctly and its results are reproducible
 - the problem is that each of islands sends its best individuals and after that they are asking for synchronization. But the server checks only whether all of the islands are already waiting for synchronization, so we cannot be sure that migration ran through correctly, and the individuals are waiting in target mailboxes. In fact, they weren't there at the time many times, so that the target island had nothing to immigrate. This behaviour caused that after some time there appeared twice more immigrants than expected. (in many cases this situation caused deleting the whole population on the island)
 - this problem is solved this way:
 - each of islands sends its best individuals
 - asks for synchronization
 - while in the mailbox there are no immigrants:
 - checks if the exchanger wants to quit
 - try to immigrate individuals
 - continue

EvolutionState:

- ***quit-on-run-complete-from-number-of-evaluations-exceeded*** (*base = „ec“*)
 - *boolean*
 - specifies whether to shut down the entire evolutionary run when specified number of evaluations was exceeded
 - note.: this parameter is independent on *quit-on-run-complete* parameter

VectorSpecies:

- ***mutation-prob*** (*base = ec.vector.VectorSpecies*)
 - *float* (*in basic*)
 - *String* (*added*)

- This parameter is specified as string „1/genome-length“, then is mutation automatically computed as 1/genomeSize in VectorSpecies initialization and the result is printed out on standard output

B.2. Plotting and Exporting

- **printBest** (base = ec.my.Source.util.myDisp)
 - my plotting function which plots two lines: best (red) and average (blue) fitness in each generation
 - example of use:
 - stat.num-children = 1
 - stat.child.0 = ec.my.Source.util.myDisp.PrintBest
 - function is incomplete and should be rewritten :-)
- **print-best** (base = ec.my.Source.util.myDisp)
 - *boolean*
 - whether to print out line with best fitness values
 - default value is *true*
- **print-mean** (base = ec.my.Source.util.myDisp)
 - *boolean*
 - whether to print out line with mean fitness value
 - default value is *true*
- **dynamic-x-axis** (base = ec.my.Source.util.myDisp)
 - *boolean*
 - whether to X-axis dynamically adapt to range of generations number
 - *true*: at start is on X zero, and range increases with num. of generations
- **dynamic-y-axis** (base = ec.my.Source.util.myDisp)
 - *boolean*
 - whether to Y-axis dynamically adapt to range of Y values(fitness vals.)
 - *true*: all the time of run, Y axis range is set exactly to range of Y vars.
- **lock-y-axis-to-borders** (base = ec.my.Source.util.myDisp)
 - *boolean*
 - whether to Y-axis automatically adapt to maximum possible range of fitness values
 - these data are stored in myFitness.min/max
 - if !myFitness.bordersDefined, and axis is not dynamic, its range will be automatically set to <0,1>
- **lock-y-max-to** (base = ec.my.Source.util.myDisp)
 - *double*
 - manually defines the maximum value on Y axis
- **lock-y-min-to** (base = ec.my.Source.util.myDisp)
 - *double*
 - manually defines the minimum value on Y axis
 - **note**: graphs have ability to automatically enlarge even over specified borders, although it shouldn't be necessary..
- **lock-y** (base = ec.my.Source.util.myDisp)
 - *boolean*
 - we can define by this, that we want to lock y axis boundaries to initial values (using for example *lock-y-min-to*)
 - if true, axis y is not adapted to any of range changes

- **wait** (base = ec.my.Source.util.myDisp)
 - *int*
 - defines how many miliseconds to wait after printing-out the actual graph
 - warning: this parameter forces to wait entire island, so if asynchronous communication it changes conditions of entire evolutionary run
- **counter** (base = ec.my.Source.util.myDisp)
 - *int*
 - defines how many generations to wait (from start of run) with y axis borders set to dynamic
 - so it dynamically adapts **counter** generations
- **round-function** (base = ec.my.Source.util.myDisp)
 - *int*
 - it switches between rounding functions (to y axis range)
 - at this moment it has range <0,4> where number 4 is best by then, but this is not complete and 100% stable unfortunately..
 - number 3 = doNotAdapt()
- **print-my-fitness** (base = ec.my.Source.util.myDisp)
 - *boolean*
 - if *state.myFitness.useMyFitness=true*, we can select whether to print out *myFitness*, or *Fitness*
 - default: *printMyFitness = state.myFitness.useMyFitness*
- **my.name** (base = „ “)
 - *String*
 - names of particular islands in the graphs
- **printOsmera** (base = ec.my.Source.util.myDisp)
 - two lines: Osmera's function (blue) and the best individual in each generation
 - remain is the same as *printBest*

myMatlabStatistics:

- **export-original-fitness** (base = ec.my.Source.util.myMatlabStatistics)
 - *boolean*
 - if *myFitness* is used (*state.myFitness.useMyFitness=true*) we can specify if we want export also these values (best and *mean Fitness*)
 - placed in **Problem folder**
- **gzip** (base = ec.my.Source.util.myMatlabStatistics)
 - *boolean*
 - whether to compress results
- **file** (base = ec.my.Source.util.myMatlabStatistics)
 - *String*
 - the name of file to create
 - placed in **Apps folder**
- **gather-full** (base = ec.my.Source.util.myMatlabStatistics)
 - *boolean*
 - extended statistics about times and so on

B.3. Task parameters

Knapsack Problem:

- **oscilating** (*base = ec.my.Source.Knapsack*)
 - *boolean*
 - whether this problem will be dynamic or not
- **static-generations** (*base = ec.my.Source.Knapsack*)
 - *int*
 - how often to switch between the targets [generations]
- **target** (*base = ec.my.Source.Knapsack*)
 - *String <1,0>*
 - specification of target solution
 - simply write f.e.: „1010101...1“ of length *n*, where *n* is genome-size
- **target2** (*base = ec.my.Source.Knapsack*)
 - *String <1,0>*
 - optional second target solution (if *oscilating = true*)
- **fitness.normalize** (*base = ec.my.Source.Knapsack*)
 - *boolean*
 - whether to normalise fitness betwen <0,1>

Osmera Problem:

- **Osmera** (*base = ec.my.Source.Osmera*)
 - implementation of Osmera's dynamic problem

Real Functions:

- **RealFunction** (*base = ec.my.Source.RealFunction*)
- **lower-bound** (*base = ec.my.Source.RealFunction*)
 - *double*
 - definition of lower bound for input variables
- **upper-bound** (*base = ec.my.Source.RealFunction*)
 - *double*
 - definition of upper bound for input variables
- **number-of-variables** (*base = ec.my.Source.RealFunction*)
 - *int*
 - how many input variables will the real function have
 - *genome-size % number-of-variables* must be zero
- **tolerance** (*base = ec.my.Source.RealFunction*)
 - *double*
 - tolerance from ideal fitness to be considered as ideal individual
- **name** (*base = ec.my.Source.RealFunction*)
 - name of the concrete real function fol solving
 - the specified function must be child of *AbstractRealFunction*
 - example of use:
 - `eval.problem.name = ec.my.Source.util.Rastrigin`

- listing of implemented real functions:
 - *Rastrigin*
 - *Sphere*
 - *Schwefel*
 - *Rosenbrock*
- *use-my-fitness* (*base = ec.my.Source.RealFunction*)
 - *boolean*
 - whether to use myFitness with original values of fitness function (as defined in [10] , not transformed to positive maximised fitness)
 - default = false – as default is transformed fitness (graphs,txt exports..)

RoyalRoad:

- *part_length* (*base = ec.my.Source.RoyalRoad*)
 - *int*
 - length of one RoyalRoad part [in bits] from definition
- *number_of_parts* (*base = ec.my.Source.RoyalRoad*)
 - *int*
 - how many parts will RoyalRoad contain
 - its rather for control, because *part_length*number_of_parts* has to be equal to *genome-size*
- *part_cost* (*base = ec.my.Source.RoyalRoad*)
 - *int*
 - how much to add to fitness if the part contains only ones
 - it is optional parameter, by default: *part_cost = part_length*
- *fitness.normalize* (*base = ec.my.Source.RoyalRoad*)
 - *boolean*
 - whether to normalise fitness between <0,1>

HIFF:

- *k* (*base = ec.my.Source.HIFF*)
 - *int*
 - branching coefficient (how many branches from one)
 - it's optional parameter, by default *k=2*
- *p* (*base = ec.my.Source.HIFF*)
 - *int*
 - depth of the HIFF tree
 - it must be equal to: *k-th sqrt(genome.size)* , where *k* is branching coefficient
- *rc* (*base = ec.my.Source.HIFF*)
 - *int*
 - contribution to fitness
 - contribution of element in depth *x* will be: rc^x (*x* is from: <0,*k*>)
 - it's optional parameter and by default *rc=2*

Attachment C: ECJ found bugs

There was several of small and harmless bugs in the ECJ, which can be forgotten, but this chapter written because of comparatively important one, mainly in the application such is this. Many of those smaller ones I've unfortunately forgot so I will mention here only some of them. But as usual, one of the question is whether these mistakes were present before my upgrading of ECJ or not..

SimpleBreeder

The ECJ was unable to solve the dynamic problems, with breed-elites set to value bigger than 0, correctly. Because the SimpleBreeder is working in the following manner:

- Initialize the new population
- Copy the elites into the new population
- Breed, until the rest of the new population is filled

After the breeding process the *Evaluated* flag was set to false, so the SimpleEvaluator made the new evaluation of each individual. But the elites weren't bred and so they weren't reevaluated. But when the dynamic problem changed its optimum, these (probable) ideal individuals became worse.

This particular „problem“ was solved simply by indicating of dynamic problem in the parameter files, when the problem is marked as a dynamic one, the elites are also reevaluated every generation.

IslandExchange - RunComplete

There was small bug in a detecing whether the run is complete. There was the sequence written:

- TRY to read the message from the server
- Check whether the message was already accepted and the server is down already
- CATCH exception and exit (this is caught when the server is down for instance)

In cases when the server was down already, the second line of this function was unreachable, so the evolutionary run on the particular islands continued in running and writing warnings about weird communication, even in case that the server was turned off already.

Island Exchange – synchronous communication

The biggest found bug in the ECJ was this. It was occuring only in the case of synchronous communication and only from time to time. The problem is that the particular islands are running on different processes, even on each of the islands there

is running second thread called Mailbox (or even third thread called Server). This mailbox has simple purpose – wait for immigrants and store them. So we can see that this implementation of island model is totally asynchronous. When trying to communicate asynchronously, there is no problem and one of generations looks as follows:

1. Evaluate
2. Check whether to send some individuals
 - i.e. when the generation modulo migration interval == 0
 - if yes, send the individuals
 - if no continue
3. Check whether to quit
4. Breed the population
5. Evaluate the population
6. Try to receive some individuals
 - i.e. just check whether some immigrants are sitting in the mailbox
 - if yes, immigrate them
 - if no, continue
7. generation ++

Simple and nice, but the synchronous model has the different functionality:

1. Evaluate
2. Check whether to send some individuals
 - if yes:
 - send the individuals to other islands
 - if not:
 - continue
3. Check whether to quit
4. Breed the population
5. Evaluate the population
6. Check whether to receive some individuals
 - if yes:
 - send the synchronization request to the server
 - wait for the OKAY message (telling that all of the islands have synchronized)
 - when the message came continue:
 - immigrate the individuals, i.e.:
 - look at the mailbox, if there are immigrants, import them
 - if there is nothing, continue
7. generation ++

This seems to work properly too, but the problem is as follows: try to imagine this situation with two islands named A and B. The island B is the target island for the A. There is number of migrants set to the 50% of population and a migration interval is 20 generations, then the sequence is:

- A – sends the individuals
- B - sends the individuals
- A – asks for synchronization

- B – asks for synchronization
- Server - accepts all of the synchronization requests, everything is alright, sends the OKAY message
- The island B is trying to receive new individuals, but there's nothing in the mailbox, so the island continues its evolutionary run. (the problem here is, that the immigrants were sent by the island A, but they haven't reached the target Mailbox on the island B yet)
-
- generation ++
-
- .. immigrants just arrived to the target mailbox in the island B
-
-
- now the same situation here, both of islands requested the synchronization
- server is answering with the OKAY message (in this case, the immigrants from the island A came at the time)
- Island B checks whether to immigrate some individuals, the answer is yes, there is 2x50% of its population size in the mailbox, the island is immigrating until the entire population is deleted.

And this is very unwanted feature. Because the communication is synchronous, but the transfer of individuals is still asynchronous and unchecked.

I've solved this by adding the parameter: *force-receive* (look at the implementation notes/parameters) which forces an island to wait for some new immigrants after the synchronization. When there is nothing in the mailbox, the island sleeps a while and then retries to receive some immigrants. This is repeated for some time, after that aborted and evolutionary run continues with warning written out. This problem is not completely solved, but it is functionable pretty well. (i.e. when the source island ends its evolutionary run because of exceed the maximum of generations, just turns itself off, but the the target island may wait for its immigrants some specified time, until it is allowed to continue.)

Note: so in the CGA topologies, where the island A is not supposed to receive any individuals, there is parameter *force-receive* set to false. In all of the common islands, there is *force-receive* parameter set to *true* by default. This parameter works only in the synchronous communication.

Attachment D: Experiments & Modifications

On the enclosed CD containing all of the used material in this thesis, there is folder called "Bakalarka", which is a Netbeans project folder containing all the source code, parameter files and the scripts used for launching particular experiments.

All of the new (not modified) stuff is added directly into the ECJ library folder from many reasons, so the most interesting folder should be placed here: **Bakalarka/ecj/ec/my/**. It contains several subfolders and here is their purpose:

- **Source** - Contains all of the added java classes (i.e. selection methods, breeding pipelines, implementation of functions etc.)
 - o What is important: this and the other folders folder MUST contain also the actual *.class files. This couldn't be obvious, because the Netbeans are compiling the java classes to the folder: Bakalarka/build/classes/. So if you modified some source code and you want to launch the evolutionary run, you should copy the newly generated *.class file to the appropriate location. (The provided scripts copies some specified subset of *.class to it's appropriate place automatically.)
- **Alignment** - Here are parameter files storing the information about an arrangement of online graphs printed out during the evolutionary run
- **Problem** - Parameter files storing the configuration of particular implemented problems
- **Topology** - Parameter files specifying the configuration of various island topologies
- **Apps** - Parameter files for particular islands, every evolutionary run has it's own parameter files for each of the islands, the *.stat files containing the results generated during an evolutionary run are also located here
- **Scripts** - Here are the *.command files used for launching particular evolutionary run (on Mac OS X)
- **Tests** -And finally in this folder there are configurations for all of the experiments mentioned in this bachelor thesis stored, including the launching scripts and some of the generated *.stat files

All of the generated *.stat files sorted by the performed experiments are in the other folder called "Matlab".

D.1. Launching the Experiments

The ECJ defines the hierarchical tree of *.params files for configuring the evolutionary run. I divided these files to three types: *problem*, *island* and *topology*. Each of the *islands* has its parent file from the *problem* domain, and one of the islands in the *topology* has also parent from the *topology* domain, this island is also the server. The main problem was that each of the islands is running on the different process, so the ECJ's main class had to be launched for each of them. I solved this by the small system of *.command files. Each *.command file then launches the entire evolutionary run. These files are the special case of shell script files known from the OS Linux. I hope that these files could be used also under this OS, despite the fact that they are proposed to work under Mac OS X.

Each of the scripts:

- copies necessary *.class files into the proper position
- opens the needed number of new terminal windows
- in each terminal window loads all of necessary java libraries
- in each terminal window launches the particular island
 - (Using the following command: “java ec.Evolve -file \$1” ; where the “\$1” represents the name of parameter file from the *island* domain)
 - So you can load any of the topologies also manually on an arbitrary operating system.

Note: some other additional scripts are placed in: `Bakalarka/ecj/start/StartScripts/`.

For more information about this please check the ECJ tutorials in [15].

Used Abbreviations

EC	-	Evolutionary Computation
ES	-	Evolution Strategy
GA	-	Genetic Algorithm
GP	-	Genetic Programming
SGA	-	Simple Genetic Algorithm
CEA	-	Cascade Evolutionary Algorithm
CGA	-	Cascade Genetic Algorithm
PGA	-	Parallel Genetic Algorithm
DF3	-	deceptive function number 3
PDF	-	partially deceptive unction
EM	-	Evolution Model
HIFF	-	Hierarchical if and only if problem
ECJ	-	Evolutionary Computation Research System
SGT	-	Scientific Graphics Toolkit
*.params file	-	a file containing the configuration of evolutionary run
*.stat file	-	a file containing statistics about the evolutionary run
*.comand file	-	a shell script file used for launching the experiments

Bibliography

- [1] Wikipedia: http://en.wikipedia.org/wiki/Genetic_algorithm
- [2] M. Dianati, I. Song, and M. Treiber: *An Introduction to Genetic Algorithms and Evolution Strategies*, University of Waterloo, Ontario, Canada, <http://www.swen.uwaterloo.ca/~mdianati/reports/gaes.pdf>
- [3] J. Kubalík: *slides for subject X33SCP* lectured on Czech Technical University in Prague, <http://ida.felk.cvut.cz/moodle/login/index.php> , 2008.
- [4] J.H. Holland: http://en.wikipedia.org/wiki/John_Henry_Holland
- [5] P. Pošík: *Parallel Genetic Algorithms*, diploma thesis written on Czech Technical University in Prague, 2001.
- [6] Z. Michalewicz: *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag Berlin Heidelberg New York, 1994.
- [7] M. Tomassini: *Spatially Structured Evolutionary Algorithms, Artificial Evolution in Space and Time*, Natural Computing Series, Springer Berlin Heidelberg New York, 2005.
- [8] M. Horváth: *Evolutionary Design Of Logic Circuits*, bachelor thesis written on Czech Technical University in Prague, 2007.
- [9] Wikipedia: <http://en.wikipedia.org/wiki/Panmixia>
- [10] GEATbx: *Example Functions*: <http://www.geatbx.com/docu/fcnindex-01.html>
- [11] *Holland's Schema Theorem*:
<http://www.cse.unr.edu/~sushil/class/gas/notes/GASchemaTheorem2.pdf>
- [12] *HIFF, GA test function*: <http://www.cs.brandeis.edu/~richardw/hiff.html>
- [13] M. Mitchell and S. Forrest: *Fitness Landscapes: Royal Road Functions*. In Back Fogel, and Michalewicz (Eds.) *Handbook of Evolutionary Computation*. Institute of Physics Publishing, Philadelphia and Bristol UK, B2.7:1-25, 1997.
- [14] J.I. van Hemert, C. Van Hoyweghen, E. Lukschndl and K. Verbeeck: *A "Futurist" approach to dynamic environments*, Proceedings of the Workshops at the Genetic and Evolutionary Computation Conference, Dynamic Optimization Problems, pages 35-38, 2001.
- [15] S. Luke and collective: *ECJ 18, A Java-based Evolutionary Computation Research System*, <http://cs.gmu.edu/~eclab/projects/ecj/>
- [16] D.W. Denbo: *SGT 3.0, Scientific, Graphics Toolkit*, <http://www.epic.noaa.gov/java/sgt/>

- [17] J. Eggermont and T. Lenaerts: *Non-stationary function optimization using evolutionary algorithms with a case-based memory*. Tech. rep. TR-2001-11, Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands, 2001.
- [18] J. LEWIS, E. HART AND G. RITCHIE: *A comparison of dominance mechanisms and simple mutation on non-stationary problems*. In Eiben, A.E. et al. (Eds.). Proceedings of PPSN'98, LNCS 1498, pages 139-148. Springer, 1998.
- [19] P. Ošmera, V. Kvasniška and J. Pospíchal: *Genetic algorithms with diploid chromosomes*. In Proceedings of Mendel'97, pp. 111-116, 1997.
- [20] R.A. Watson, G.S. Hornby and J.B. Pollack: *Modeling Building-Block Interdependency*. In proceedings of Fifth International Conference PPSN V, pp. 97-106. Springer, 1998.