

CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF ELECTRICAL ENGINEERING



BACHELOR'S THESIS

Binary local optimizer with linkage learning

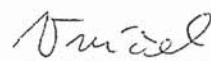
Prague, 2010

Author: Stanislav Vaníček

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v příloženém seznamu.

V Praze dne 27.5.2010



podpis

Acknowledgements

I would like to thank Ing. Petr Pošík Ph.D., my advisor. Petr was always there to listen and to give advice. Without his encouragement and constant guidance, I could not have finished this work. A special thanks goes to my family. They've still believed in me and gave me strength when I was down. Thank you!!!!

Abstrakt

V této práci popíši algoritmus na řešení optimalizačních úloh, tyto úlohy budou typu Black-box. Funkčnost algoritmu poté otestujeme na několika testovacích funkcích. Ty budou dvojího typu, aditivně dekomponovatelné funkce a hierarchické funkce. Testovat se bude počet evaluací potřebných ke spolehlivému nalezení globálního optima. Výsledky budou poté porovnány proti dvěma genetickým algoritmům ECGA a BOA.

Abstract

In this paper I will describe algorithm for Black-box optimization problems. The algorithm will be tested against two types of test functions, additively decomposable functions and hierarchical functions. Subject to testing, the number of evaluations needed to reliably find the global optima. Results will be compared with results of two genetic algorithms ECGA and BOA.

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Stanislav Vaníček
Studijní program: Softwarové technologie a management
Obor: Inteligentní systémy
Název tématu: Binární lokální optimalizátor s detekcí závislostí mezi proměnnými

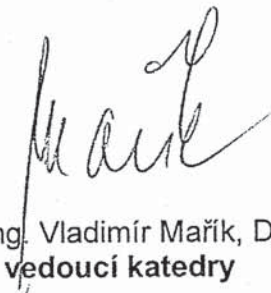
Pokyny pro vypracování:

1. Prostudujte perturbační techniky pro učení závislostí mezi proměnnými.
2. Ve zvoleném programovacím jazyce vytvořte restartovaný lokální optimalizátor, který bude schopen se za běhu učit závislostí mezi proměnnými a využívat je při prohledávání.
3. Výsledný algoritmus porovnejte na referenčních úlohách s dalšími metodami (restartovaný lokální optimalizátor bez detekce závislostí, algoritmus BOA a algoritmus ECGA), výsledky porovnejte a vyhodnoťte.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí bakalářské práce: Ing. Petr Pošík, Ph.D.

Platnost zadání: do konce zimního semestru 2010/2011


prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry




doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 30. 11. 2009

BACHELOR PROJECT ASSIGNMENT

Student: Stanislav Vaníček
Study programme: Software Engineering and Management
Specialisation: Intelligent Systems
Title of Bachelor Project: Binary Local Optimizer with Linkage Learning

Guidelines:

1. Learn about perturbation techniques for linkage learning.
2. In chosen programming language, create a restarted local optimizer with the ability to learn the linkage during the run and use it during the search.
3. Evaluate the resulting algorithm on selected set of benchmark functions and compare it to other algorithms (restarted local optimizer with no linkage learning, algorithms BOA and ECGA).

Bibliography/Sources: Will be provided by the supervisor.

Bachelor Project Supervisor: Ing. Petr Pošík, Ph.D.

Valid until: the end of the winter semester of academic year 2010/2011.


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




doc. Ing. Boris Šimák, CSc.
Head

Contents

1	Introduction	1
1.1	Optimization and optimization problems	1
1.2	Black-box optimization	1
1.3	Ways of solution	2
1.4	Chosen solution	2
1.5	Thesis organization	3
2	Dependencies between variables and their detection	4
2.1	Additively separable functions	4
2.1.1	One Max	5
2.1.2	Trap	5
2.1.3	Equal Pairs	6
2.1.4	Sliding XOR	7
2.2	Hierarchical functions	7
2.2.1	Hierarchical if-and-only-if (HIFF)	8
2.3	Linkage learning	9
2.3.1	LINC (Linkage Identification by Nonlinearity Check)	10
2.3.2	LIMD (Linkage Identification by non-Monotonicity Detection)	11
2.3.3	LIEM (Linkage Identification with Epistasis Measures)	12
2.3.4	LIEM² (LIEM considering Monotonicity)	14
2.4	Algorithms ECGA and BOA	15
2.4.1	ECGA algorithm	15
2.4.2	BOA algorithm	15

3	Local optimizer with dependencies detection	16
3.1	Local Optimizer	17
4	Experiments	19
4.1	Experiment entity	19
4.2	ECGA and BOA results	19
4.3	Graphs of convergence	20
4.4	Scalability graphs	22
4.4.1	OneMax function	22
4.4.2	Trap function	22
4.4.3	Equal Pairs function	23
4.4.4	SlidingXOR function	23
4.5	Discussion	25
5	Conclusion	26
5.1	Results	27
	Literatura	29
A	Content provided CD	I

Chapter 1

Introduction

1.1 Optimization and optimization problems

In computer science, optimization refers to choosing the best element from set of available elements.

This means solving problems which one seeks to minimize or maximize, that is our goal in this paper, a real function by systematically choosing the values of real or integer variables from within an allowed set. This formulation, using a scalar, real-valued objective function, is the simplest example. More generally, it means finding best available values of some objective function given a defined domain, including a variety of different types of objective functions and different types of domains.

1.2 Black-box optimization

In black-box optimization the objective function is a black box, that means we know nothing about it. More specific it means:

- we don't know nothing about the function derivation
- we don't know how is defined a fitness function

- we don't have the applicable implementation of the right optimization method
- we know nothing about the right optimization method

1.3 Ways of solution

In this paper we will focus on binary represented functions, therefore we will proceed in the discrete world. We have two ways how to optimize functions which are described in Chapter 2:

Local optimization is a metaheuristic for solving computationally hard optimization problems. Local search algorithms move from solution to solution in the space of candidate solutions until a solution deemed optimal is found or a time bound is elapsed.

Genetic algorithms is a search technique to find exact or approximate solutions to optimization and search problems. Genetic algorithms are categorized as global search heuristics. Genetic algorithms are a particular class of evolutionary algorithms (EA) that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover.

But there is a problem, what if there are dependencies between variables (bits) representing a solution. We need an information about groups of bits, which are dependent, if we want to find the global optimum of a given function. If we don't know anything about the dependencies and work with the variables (bits) like independent ones (one by one), it's almost impossible to find the global optimum.

1.4 Chosen solution

Solution described in this paper combines two algorithms. First part is algorithm for dependencies detection. There are many algorithms, which can give us information about dependen-

cies between variables, they are described below in Chapter 2.

The second part of solution is a local optimization. This algorithm can use information given from the algorithm for dependencies detection. So, it works over groups of bits.

1.5 Thesis organization

In chapter 2 are described dependencies between variables and algorithms to its detection. Chapter 3 describes the chosen solution in detail. In chapter 4 are experiments with the chosen solution and confrontation to other algorithms and chapter 5 is a work analysis.

Chapter 2

Dependencies between variables and their detection

If there are dependencies between variables (bits) we can't solve a problem with common genetic algorithm. If we do, we will need a very large population to get the global optimum, we hitch in local optimum in most cases.

2.1 Additively separable functions

Additively separative functions (ASF) [1] are defined

$$f = \sum_{k=1}^K a_k f_k(S_k). \quad (2.1)$$

So, they represent a linear combination of a non-linear particular functions f_k , which are defined over subset of variables \mathbf{x} . (S_k is a subset of input variables to f_k .) If sets S_k are disjoint sets (variable on position d can be only in one set S_k), components are independent. Components do not have to be continuous, that means, dependency bits from one component could be away from each other. If sets S_k aren't disjoint, we can create relatively difficult structures.

Let's have a binary chain \mathbf{x} with a length D . We can define a lot of fitness functions over this chain. We will use following four functions in this paper.

2.1.1 One Max

A benchmark based on a maximization of *One Max* function [2] is simple. The function is unimodal and easy for each optimizer. The value of this function is number of ones in a chain, so

$$f_{DbitOneMax}(\mathbf{x}) = \sum_{d=1}^D x_d \quad (2.2)$$

The minimum of this function is 0 for a chain containing only zeros, the maximum is D for a chain containing only ones. The simplicity of this function is the independence of all bits.

2.1.2 Trap

The *Trap* [2] function is made from easy *OneMax* function by simple modification. This function is very difficult for many optimizers. It's the maximization of a function

$$f_{DbitTrap}(\mathbf{x}) = \begin{cases} D, & \text{if } \mathbf{x} = \mathbf{11\dots1}, \\ D - 1 - f_{DbitOneMax}(\mathbf{x}), & \text{otherwise.} \end{cases} \quad (2.3)$$

The function *Trap* is a linear combination of all bits, same as the function *OneMax*, except one point: the maximum, we are searching for, is where we minimum await.

The complexity of this problem is an apparent independence of bits over the whole domain. But if we will look on bits independently to others, we will be caught in local optimum - mistaken attractor. Dependence between the bits we find only when sludges optimum.

*D*bit trap optimizing is wasting time. A lot of real problems can be broken down to semi-independent subtasks, which are very difficult. The *Trap* function can be easily modified like one of these problems - an interconnection of several bits into groups of independent. This function will be separable to independent very difficult subtasks. For example *Trap* from 10 5bits blocks is defined:

$$f_{10x5bitTrap}(\mathbf{x}) = \sum_{k=1}^{10} f_{5bitTrap}(x_{5(k-1)+1}, \dots, x_{5k}) \quad (2.4)$$

The definition is analogous for *Trap* from 8bits block.

2.1.3 Equal Pairs

This is a maximization task [3]. The fitness function is defined

$$f_{DbitEqualPairs}(\mathbf{x}) = 1 + \sum_{d=2}^D f_{EqualPair}(x_{d-1}, x_d), \quad (2.5)$$

where are defined individual items with length 2 (bits)

$$f_{EqualPair}(x_1, x_2) = \begin{cases} 1, & \text{if } x_1 = x_2 \\ 0, & \text{if } x_1 \neq x_2 \end{cases} \quad (2.6)$$

This function is symmetric, that means each chain and its inverse have the same evaluation. The minimum of this function is 1 and it's for chains where zeros and ones alternate. The maximum is of this function is D and it's for chains where are only ones or only zeros. There two sources of difficulty:

1. Function is bimodal, has two optima, thanks to symmetry. Chromosomes should converge towards them. But later, when one of parents will have sequence of ones and the second parent will have sequence of zeros it could create the child with less quality, that is for GA.
2. Function contains dependencies between pairs of bits: optimal bit value depends on his predecessor. These dependencies are transmitted from bit to bit, thanks to this the whole chain is one big item.

Dependencies in this function are much simpler than dependencies in *Trap*. We could optimize its basic version. But we could define separable version with k bits blocks, like by the *Trap*. For example *Equal Pairs* from 10 5bits blocks is defined:

$$f_{10x5bitEqualPairs}(\mathbf{x}) = \sum_{k=1}^{10} f_{EqualPairs}(x_{5(k-1)+1}, \dots, x_{5k}) \quad (2.7)$$

Each quintuple of bits is evaluated by function $f_{5bitEqualPairs}$. The definition is analogous for *EqualPairs* from 8bits block.

2.1.4 Sliding XOR

This is a fitness function [3] maximization task.

$$f_{DbitSlidingXOR}(\mathbf{x}) = 1 + f_{AllEqual}(\mathbf{x}) + \sum_{d=3}^D (1 - f_{XORPattern}(x_{d-2}, x_{d-1}, x_d)). \quad (2.8)$$

As $f_{EqualPair}$ function has dependencies between 2 variables, $f_{XORPattern}$ function has dependencies between 3 variables:

$$f_{XORPattern}(x_1, x_2, x_3) = \begin{cases} 1, & \text{if } x_1 \oplus x_2 = x_3 \\ 0, & \text{otherwise} \end{cases} \quad (2.9)$$

that means, a value is 1 if 3 bits make a row from XOR functions truth-table. But in this function is returned 1 if XOR function of 3 consecutive bits returns 0, because of $1 - f$. So, if chain contains triple 001, 010 or 111. Sum in theorem above returns maximum for chain 001001001001...001 or for chain 11111...111. The function

$$f_{AllEqual}(\mathbf{x}) = \begin{cases} 1, & \text{if } \mathbf{x} = \mathbf{00...0} \text{ or } \mathbf{x} = \mathbf{11...1}, \\ 0, & \text{otherwise,} \end{cases} \quad (2.10)$$

ensure, that there is only one global optimum 11...1. The maximum of function $f_{DbitSlidingXOR}$ will be equal D , thanks to adding 1. This function creates dependencies as the function above, but with ternary of bits. There is created indirect linkage between all bits in chain, because of shifting of basic function over ternary of bits. We can also create separative version of this function by implementation of short independent blocks, for example:

$$f_{10x5bitSlidingXOR}(\mathbf{x}) = \sum_{k=1}^{10} f_{5bitSlidingXOR}(x_{5(k-1)+1}, \dots, x_{5k}). \quad (2.11)$$

2.2 Hierarchical functions

The purpose of this hierarchical functions [2] is to design a class of challenging problems that can be used to test the scalability of optimization algorithms on difficult hierarchical problems. In this section are presented two types of hierarchical trap functions.

2.2.1 Hierarchical if-and-only-if (HIFF)

The hierarchical if-and-only-if function [5] was proposed as an example of a function that is not separable and should therefore challenge even those GAs that are capable of finding building blocks of bounded order.

The structure of HIFF is a balanced binary tree. The input to the contribution¹ and mapping² functions therefore consists of two symbols. A single mapping function is used on all levels where 00 is mapped into 0, 11 is mapped into 1 and everything else is mapped into the null symbol '-'. On each level, blocks 00 and 11 contribute to the overall fitness by 2^{level} , where *level* is the number of the current level. Anything else doesn't contribute to the overall fitness. Each single bit in the tree contributes to the fitness by 1. Since the structure is a balanced binary tree, the size of the problem should be a power of 2. Figure 2.1 shows the three HDF components defining HIFF.

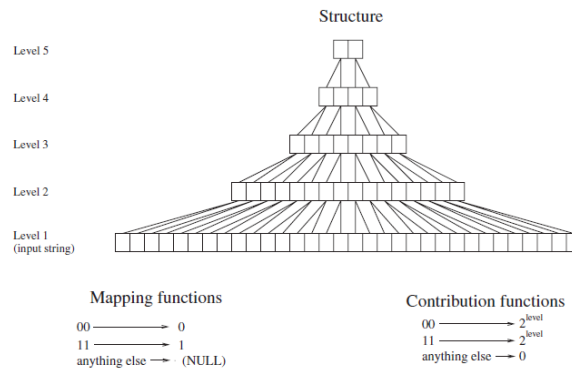


Figure 2.1: The tree components defining a 32-bit HIFF function

HIFF has two global optima, one in the string of all ones and one in the string of all zeros. An optimizer must preserve either zeros or ones on all string positions to ensure that the optimum can be reached. There are two ways of solving HIFF. The algorithm can decide whether to go after zeros or ones, or preserve both alternatives as the optimization proceeds. In the second

¹The function determines how the different blocks involved in the total value of fitness function

²A mathematical expression relating observed recombination fraction to map distance expressed in centiMorgans (a unit of recombinant frequency for measuring genetic linkage).

case, the algorithm must ensure conservation of the partitions on the current level of optimization, because mixing zeros with ones moves the optimization one or more levels down. The pieces of zeros and ones must be combined together effectively.

2.3 Linkage learning

To identify linkage groups, several algorithms were proposed. They are classified [2] into three categories:

1. Direct detection of bias in probability distribution
2. Direct detection of fitness changes by perturbation
3. Direct detection along genetic search of BBs

For the first category, several algorithms such as the estimation of distribution algorithm (EDA) (Mühlenbein & Paaß, 1996), the univariate marginal distribution algorithm (UMDA) (Mühlenbein, 1997), the factorized distribution algorithm (FDA) (Mühlenbein & Mahnig, 1999), the bivariate marginal distribution algorithm (BMDA) (Pelikan & Mühlenbein, 1999), and the Bayesian optimization algorithm (BOA) (Pelikan, Cantú-Paz, & Goldberg, 1998) were proposed to identify linkage groups by detecting bias on probability distributions after selections.

For the third category, the linkage learning GA (LLGA) (Harik, 1997) employs a two-point like crossover over circular strings to grow tight linkages of BBs. The LLGA works effectively on problems with exponentially scaled subfunctions, but fails to exploit linkage groups in uniformly-scaled problems. This is because simultaneous search for linkage groups and BBs may cause a negative feedback effect that prevents each other from obtaining correct results.

In this thesis I will concentrate on the second group. In the following, I will describe four algorithms for linkage detection using perturbation.

2.3.1 LINC (Linkage Identification by Nonlinearity Check)

LINC [6] is an effective method for dividing a larger problem into small sub-problems. For example, if we need to solve a maximization problem for a function that is represented as a sum of two independent partial functions, maximizing each partial function independently is more efficiently method of solution. In LINC, division is done in terms of Building Blocks (BB's). BB's are generated as an approximated best solution with maximum linkage value inside each linkage set. These generated BB's are combined to get a good final result.

For example, lets consider a function $f(x) = \sum_{n=1}^N f_n(x_n)$ with Linear sub functions. The character string s is created by joining all the encoded bit strings x_n that are the encoded bit strings, respectively (Figure 2.1).

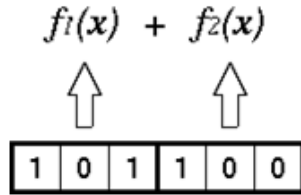


Figure 2.2: Example of function

Here $\Delta f_i(s)$ represents the fitness value change when i^{th} of string s is perturbed. Similarly $\Delta f_{ij}(s)$ represents the fitness value change when both the i^{th} & j^{th} bit of string s are perturbed.

Therefore if we consider a string $s = s_1s_2s_3s_4s_5\dots s_m$ and define changes of fitness values by bit-wise perturbations to s as follows.

$$\Delta f_i(s) = f(\dots\bar{s}_i\dots) - f(\dots s_i\dots) \quad (2.12)$$

$$\Delta f_j(s) = f(\dots\bar{s}_j\dots) - f(\dots s_j\dots) \quad (2.13)$$

$$\Delta f_{ij}(s) = f(\dots\bar{s}_i\bar{s}_j\dots) - f(\dots s_i s_j\dots) \quad (2.14)$$

Where, $f(s)$ is fitness of individual s , and $\bar{s} = 1 - s_i$ (Means that change $0 \rightarrow 1$ and $1 \rightarrow 0$) s_i represents the i^{th} bit of string s .

If $|\Delta f_{ij}(s) - \Delta f_i(s)| > e$, that is, changes of fitness values by perturbations on s_i and s_j are additive, which indicates a linear interaction between them. However, if $\Delta f_{ij}(s) \neq \Delta f_i(s) + \Delta f_j(s)$, this means that they are not additive, which simply means nonlinearity. Checking non linearity in only one string is not enough because there may exist linearity inside a BB in some contexts. Therefore, all the strings in a properly sized population must be checked. If linearity is detected for all the string in a pair of loci than it is safe to keep them as unlinked.

To store linkage groups, we assign a linkage set - a list of loci which are tightly linked - to each locus, concluding the above explanation.

1. If $\Delta f_{ij}(s) \neq \Delta f_i(s) + \Delta f_j(s)$ then s_i and s_j are surely members of a linkage set, so we add i to the linkage set of locus j and add j to the linkage set of locus i . Direct detection of fitness changes by perturbation.
2. If $\Delta f_{ij}(s) = \Delta f_i(s) + \Delta f_j(s)$, the s_i and s_j may not be a member of a linkage set, or they are linked but linearity exists in the current context we do nothing in this case.

We can introduce the value “ e ” that specifies the amount of effort allowed for linearity/non linearity detection and replace the above $\Delta f_{ij}(s) \neq \Delta f_i(s) + \Delta f_j(s)$ by $|\Delta f_{ij}(s) - \Delta f_i(s) - \Delta f_j(s)| > e$.

2.3.2 LIMD (Linkage Identification by non-Monotonicity Detection)

Instead of checking non linearity like in LINC procedure, the Linkage Identification by non-Monotonicity Detection (LIMD) [6] procedure checks the violation of monotonicity conditions to detect linkage groups.

A monotonous function and non-monotonous function are shown in Figure 2.2 respectively.

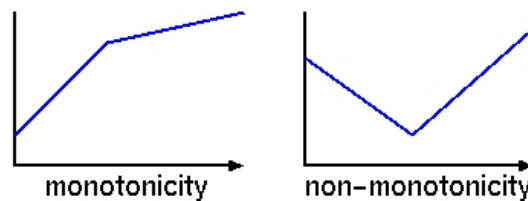


Figure 2.3: Monotonicity (left), Non-monotonicity (right)

As shown in the figure the Monotonic functions can be easily solved by using simple "Hill Climbing" methods, such problems are also easy for the GA's. In such cases even if the change of fitness shows non-linear behavior still it is possible to find the optimal solution for the problem.

The equations for monotonicity and non-monotonicity can be described by the following expressions. The procedure adds a pair of loci (i, j) to the linkage set when the following condition is not satisfied in at least one string in population.

$$\begin{aligned}
 & \text{if } (\Delta f_i(s) > 0 \text{ and } \Delta f_j(s) > 0) \\
 & \text{then } (\Delta f_{ij}(s) > \Delta f_i(s) \text{ and } \Delta f_{ij}(s) > \Delta f_j(s)) \\
 & \text{if } (\Delta f_i(s) < 0 \text{ and } \Delta f_j(s) < 0) \\
 & \text{then } (\Delta f_{ij}(s) < \Delta f_i(s) \text{ and } \Delta f_{ij}(s) < \Delta f_j(s))
 \end{aligned}$$

Where, $\Delta f_i(s)$, $\Delta f_j(s)$, $\Delta f_{ij}(s)$ are the same as defined in LINC.

2.3.3 LIEM (Linkage Identification with Epistasis Measures)

The LIEM [9] replaces the strict condition of the LINC with condition based on a linkage measure that represents strength of epistasis between loci. The linkage should be identified by detecting difference between strong epistasis and weak one. Weak epistasis among a set of loci means that the problem can be decomposed into subproblems regarding the loci and will be easily optimized separately. Beyond, a set of loci with strong epistasis are difficult to separate and optimize, therefore they should be treated all together along optimization process through recombination operators. Genetic search with recombination operators processes relatively weak epistasis and strong epistasis. A deception can only be processed with enumerative search realized in an enough-sized population of strings (the population needs to have $O(2^k)$ strings where k is the maximum order of BBs). Figure 2.3 shows this idea of linkage identification.

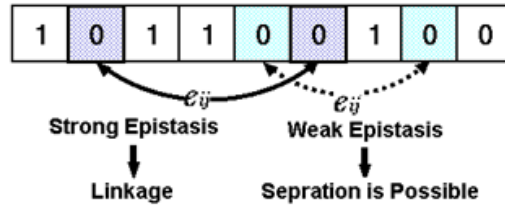


Figure 2.4: Overview of LIEM

The LIEM we propose aims to deal with linkage identification based on a clear definition of *strength of epistasis*. This approach is based on an *epistasis measure* $e_{ij} \geq 0$ defined for each pair of loci (i, j) . For example, a simple epistasis measure based on the LINC criterion can be defined as follows:

$$e_{ij} = \max_{s \in P} |\Delta f_{ij}(s) - (\Delta f_i(s) + \Delta f_j(s))|, \quad (2.15)$$

Where

$$\Delta f_i(s) = f(\dots \bar{s}_i \dots) - f(\dots s_i \dots) \quad (2.16)$$

$$\Delta f_j(s) = f(\dots \bar{s}_j \dots) - f(\dots s_j \dots) \quad (2.17)$$

$$\Delta f_{ij}(s) = f(\dots \bar{s}_i \bar{s}_j \dots) - f(\dots s_i s_j \dots), \quad (2.18)$$

$s = s_1 s_2 s_3 s_4 s_5 \dots s_m$ and $\bar{s}_i = 1 - s_i$ (Means that change $0 \rightarrow 1$ and $1 \rightarrow 0$) s_i represents the i^{th} bit of string s .

Here, we employ the above simple definition. However, we can assume another definition of linkage measure. The epistasis measure represents a *tightness of linkage* for the pair of loci. Therefore, a *linkage group* of a locus is identified by sorting epistasis measures concerning the locus and picking up a fixed number of loci k from those with larger value of the measure. For example, when we have the following values for epistasis measures e_{ij} for locus i ,

$$e_{12} = 0.5, e_{13} = 1.1, e_{14} = 0.3, e_{15} = 0.0, e_{16} = 0.1, \quad (2.19)$$

and we pick up three loci as a linkage group, first, e_{ij} are sorted as follows:

$$e_{13} > e_{12} > e_{14} > e_{16} > e_{15}, \quad (2.20)$$

and second, we pick up three loci according to the sorted e_{ij} and obtain $\{3,2,4\}$ as tightly linked with locus 1 and consequently, the obtained linkage group is $\{1,2,3,4\}$. Note that loci with no epistasis should not be included in the linkage group. In the above definition of epistasis measure, a pair of loci (i, j) do not consider to be tightly-linked when $e_{ij} = 0$.

To apply the LIEM, we need to assume the maximum length of BBs as the fixed number of loci k defined above. In this paper, we call the order *difficulty* number d because it represents the problem difficulty for genetic recombinations. An initial population of $O(2^k)$ strings becomes necessary to obtain correct linkage groups. Rather, it is more natural to argue that when the initial population size is fixed, the maximum length of BBs ? how many order of BBs can detect ? is fixed.

2.3.4 LIEM² (LIEM considering Monotonicity)

The epistasis measure of the LIEM² [6], which is based on the LIMD condition, is defined as follows:

$$e_{ij} = \max_{s \in P} g(\Delta f_{ij}(s), \Delta f_i(s), \Delta f_j(s)) \quad (2.21)$$

Where, Function $g(x, y, z)$ is defined as:

$$g(x, y, z) = \begin{cases} tr(y-x) + tr(z-x), & (y > 0, z > 0) \\ tr(x-y) + tr(x-z), & (y > 0, z > 0) \\ 0, & otherwise \leq \pi \end{cases} \quad (2.22)$$

$$tr(x) = \begin{cases} x, & (x \geq 0) \\ 0, & (x < 0) \leq \pi \end{cases} \quad (2.23)$$

This equation gives “0” in case of Monotonicity but gives the Measure of Non-Monotonicity in case the function is Non-Monotonic.

$$e_{ij} = \max_{s \in P} |\Delta f_{ij}(s) - (\Delta f_i(s) + \Delta f_j(s))|, \quad (2.24)$$

if calculated gives.

$$e_{ij} = \max_{s \in P} g(\Delta f_{ij}(s), \Delta f_i(s), \Delta f_j(s)) \quad (2.25)$$

2.4 Algorithms ECGA and BOA

ECGA and BOA algorithms we will use for comparison in this thesis. These algorithms are Estimation-of-distribution algorithms (EDA) [10]. Estimation-of-distribution algorithms are a subclass of evolutionary algorithms (EA), which belong to the *Direct detection of bias in probability distribution* group, described in Chapter 2 Section 2.3. The structure of these algorithms is:

1. Initialization and evaluation
2. Parent selection
3. Learning a probabilistic model based on the selected parents
4. Create children by sampling learned model
5. Children evaluation
6. Replacement strategy
7. Until termination condition is met, proceed to step 2

EDA algorithms are different from EA algorithms only in steps 3 and 4, where probabilistic model learning and sampling from it unlike crossing and mutation is used.

2.4.1 ECGA algorithm

ECGA algorithm [8] uses Marginal Product Model, so the algorithm detects groups of dependent bits. Each group is modeled by grouped probability distribution. Bits in different groups are considered to be independent.

2.4.2 BOA algorithm

BOA algorithm [2] uses Bayesian network as a model, a more general model than MPM in ECGA is, which uses conditional dependencies between the bits.

Chapter 3

Local optimizer with dependencies detection

The algorithm studied in this work has 2 main parts. One of the main parts is a linkage learning algorithm, rather LIMD algorithm which. The second main part is a local optimizer which can work with dependencies between variables found by LIMD.

There are two versions of the algorithm, but they are different in one thing only, in linkage learning process. The first version uses completely randomized linkage learning process. That means, LIMD is always started from a new randomly generated point. The structure of this version:

1. Initialize the dependency structure with no dependencies and fitness function
2. Run Local optimizer
3. Run LIMD from a new random point
 - If new dependency is found, then run Local optimizer from the actual point and run LIMD from a random generated point
 - If new dependency isn't found, then run LIMD from randomly generated point
4. If number of evaluations is spend, then return best so far solution

The second version tries to use some kind of heuristic. It works with solutions found by LIMD and tries to use it. The structure of the this version:

1. Initialize the dependency structure with no dependencies and fitness function
2. Run Local optimizer
3. Run LIMD
 - If new dependency is found, then run Local optimizer from the actual point and run LIMD from best so far solution over the actual point found by itself.
 - If new dependency isn't found, then run LIMD from randomly generated point
4. If number of evaluations is spend, then return best so far solution

3.1 Local Optimizer

This part of the algorithm is a classical local optimizer that means it tries all possible solutions from the neighborhood of the current one and chooses the best solution of them over the fitness function. But there is one more thing it can do, it can work with dependencies. So if LIMD returns some dependency between variables (bits), Local Optimizer works over group of variables. For example if we have chain with length four and bits on first and second position are dependent:

Original chain and 1. possibly solution : [0000]

2. possibly solution : [1000]

3. possibly solution : [0100]

4. possibly solution : [1100]

As is shown above, there are four possibilities for two linked variables. Similarly for four linked bits over same chain that means all variables are linked:

Original chain and 1. possibly solution : [0000]

2. *possibly solution* : [1000]

3. *possibly solution* : [0100]

4. *possibly solution* : [1100]

5. *possibly solution* : [0010]

6. *possibly solution* : [1010]

⋮

16. *possibly solution* : [1111]

So, there are sixteen possibilities for four linked variables. For n linked variables is there 2^n possibilities as is demonstrated in two examples above. That means number of possible solutions increases exponentially with number of linked variables.

Chapter 4

Experiments

4.1 Experiment entity

The main theme of experiments was a question, how many evaluations is needed to find global optimum. The algorithm ran 30 times and global optimum had to be found in every run. There were three algorithms in this competition, Local algorithm with linkage learning (two versions)(described in Chapter 3), ECGA algorithm (described in Chapter 2) and BOA algorithm (described in Chapter 2).

It's very hard to estimate number of evaluations needed to find global optimum for all functions described in Chapter 2. But in one case it is easy job, the function OneMax has no dependencies between bits, so only Local Optimizer is needed to find global optima, therefore $n * 2$, where n is length of chain, evaluations is needed to find global optima. In other cases I can't estimate number of evaluations needed to find global optima without any knowledge about optimising given function.

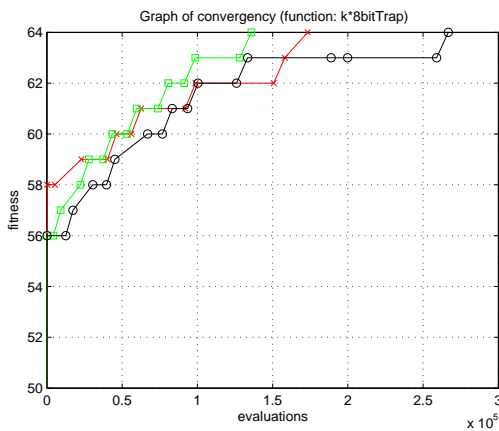
4.2 ECGA and BOA results

The results of ECGA and BOA algorithms are with the smallest possible population which can find global optima in every run of thirty. Both algorithms using tournament selection of

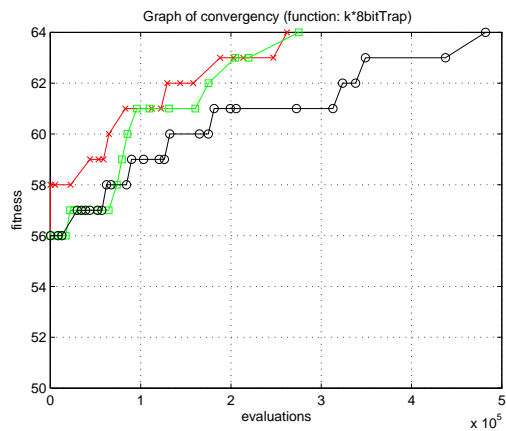
size 7 and restricted tournament replacement with window size equal to the length of strings [7]. This smallest possible population was determined by Bisection method. The Bisection method is a search algorithm for finding given value from in sorted list by shortening the list by half in each step. The Bisection method finds median, compares it with sought value and decides for top or bottom half of list because of result.

4.3 Graphs of convergence

In this section I'll present graphs of convergence on three random starting points. These graphs show how the value of the fitness function varies with the number of evaluations. Figure 4.1 shows convergence for Trap function, Figure 4.2 for EqualPairs function and Figure 4.3 for SlidingXOR function.

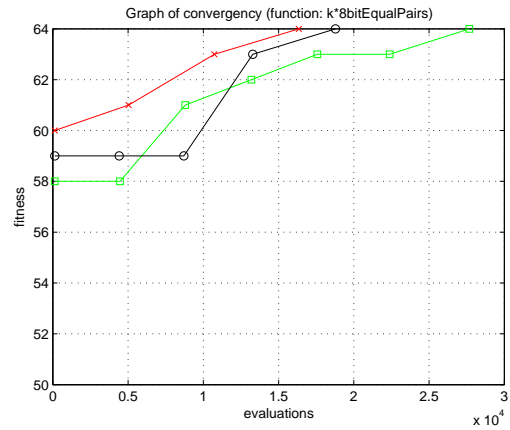
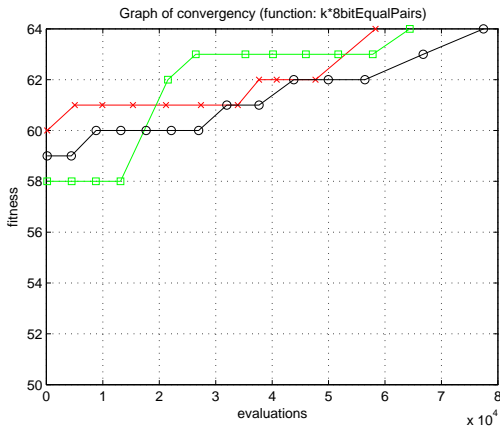


(a) 8x8bitTrap function: LIMD BSF continue version



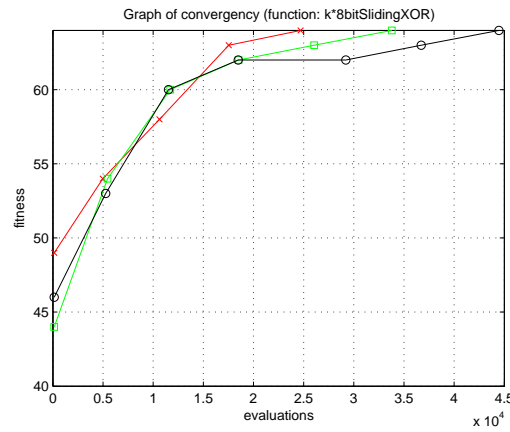
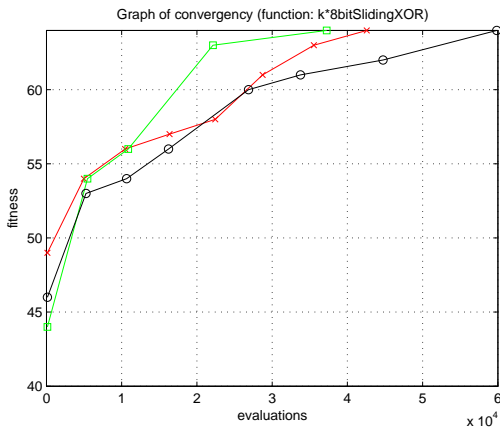
(b) 8x8bitTrap function: Randomize version

Figure 4.1: Trap function: Increasing the value of the fitness function during evaluation



(a) 8x8bitEqualPairs function: LIMD BSF continue version (b) 8x8bitEqualPairs function: Randomize version

Figure 4.2: EqualPairs function: Increasing the value of the fitness function during evaluation



(a) 8x8bitSlidingXOR function: LIMD BSF continue version (b) 8x8bitSlidingXOR function: Randomize version

Figure 4.3: SlidingXOR function: Increasing the value of the fitness function during evaluation

4.4 Scalability graphs

Scalability graphs show how many evaluations are needed to find global optimum with increasing dimension. The results are for two lengths of linkage variables group to show how number of evaluations is changed by increasing length of these groups.

4.4.1 OneMax function

The first function is DbitOneMax function. There is no need to test two different lengths of linkage learning groups because all variables are independent. As is shown on Figure 4.4 Local Optimizer with Linkage Learning (our algorithm) returns best results. These results are expected because our algorithm needs only $2 * k$ evaluations where k is length of chain, explained in Chapter 3 Section 3.1, and it may be less than genetic algorithms need to convergence.

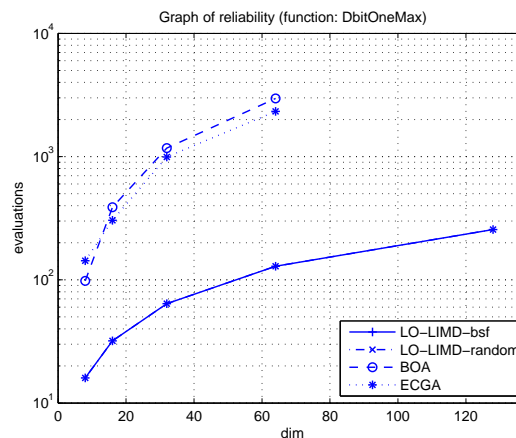
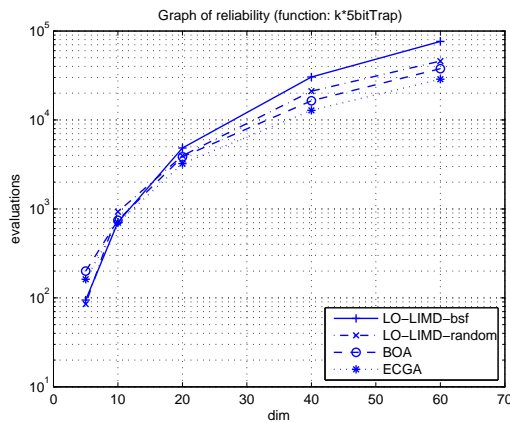


Figure 4.4: DbitOneMax function: Number of evaluations needed to find global optimum for increasing length of chain

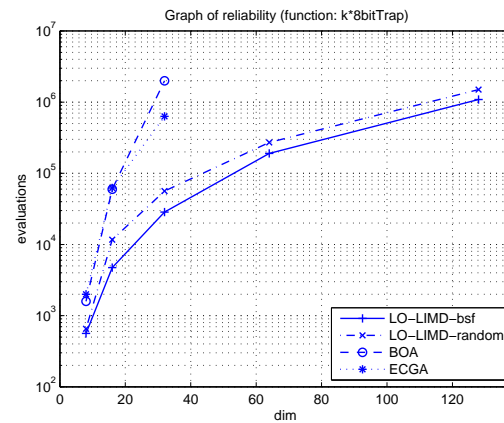
4.4.2 Trap function

For this function and for functions below, it is interesting to test two different length blocks of dependent bits. The prerequisite for this test is the longer blocks of dependent bits, the bigger number of evaluations is needed. The Trap function is the heaviest challenge for optimizers

from additive functions. Results (Figure 4.5) of this 8bit version function are really great, where genetic algorithms (ECGA and BOA) fail our algorithm continues. The results of 5bit version are comparable that means they are in the same order.



(a) kx5bitTrap function



(b) kx8bitTrap function

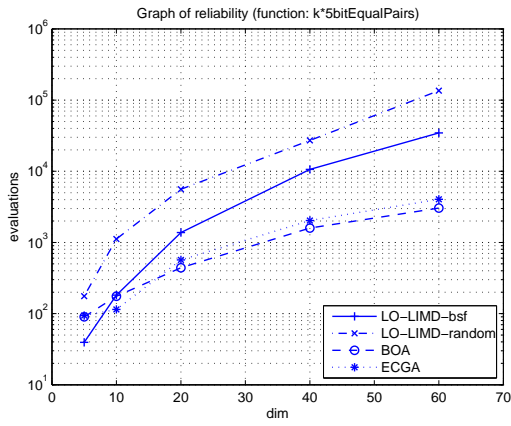
Figure 4.5: Trap function: Number of evaluations needed to find global optimum for increasing number of blocks

4.4.3 Equal Pairs function

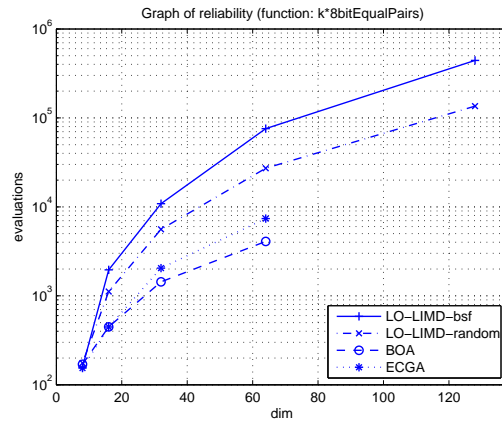
Results of this function (Figure 4.6) show that our algorithm is worse than genetic algorithms. I think, it's because it can't work with single pairs, it must work under all group.

4.4.4 SlidingXOR function

This is the last function from group of additive functions. ECGA algorithm lost to other algorithms, so it failed. Our algorithm is comparable to BOA algorithm and random version is even better than BOA in 8bit version of function. Results are shown on Figure 4.7.

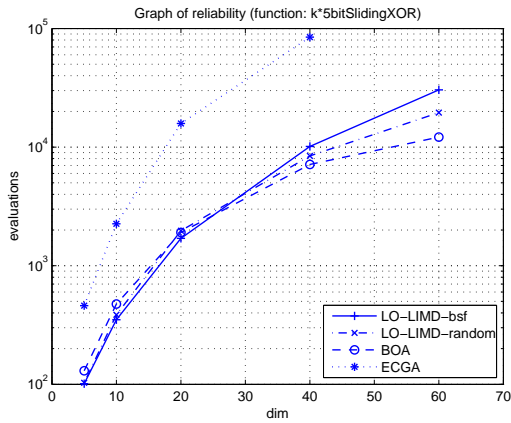


(a) $k \times 5$ bitTrap function

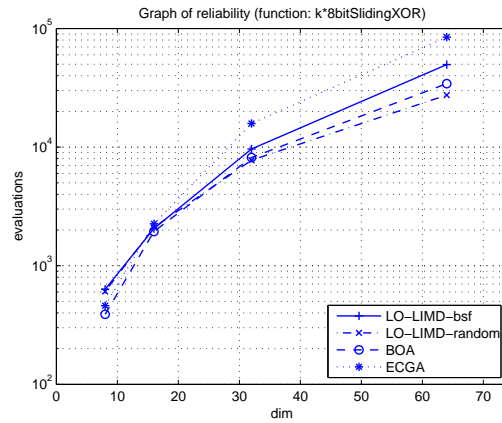


(b) $k \times 8$ bitTrap function

Figure 4.6: EqualPairs function: Number of evaluations needed to find global optimum for increasing number of blocks



(a) $k \times 5$ SlidingXOR function

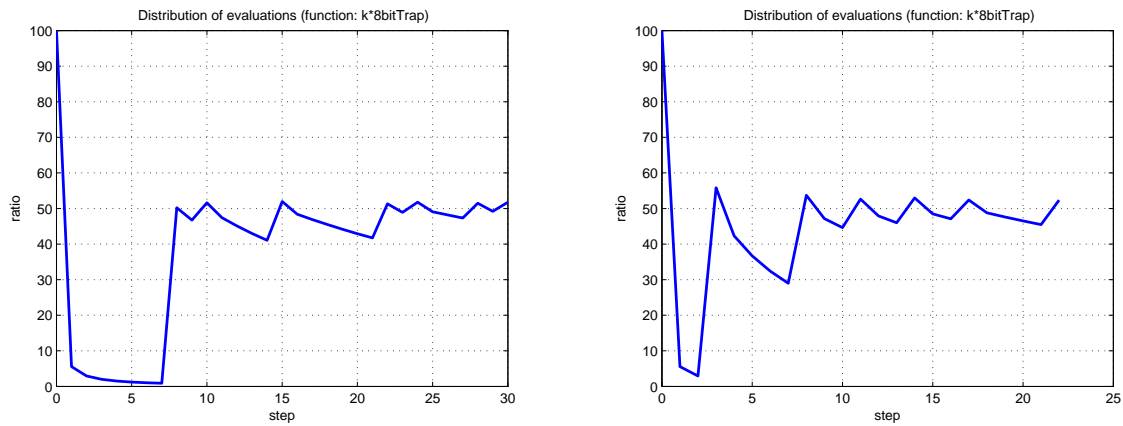


(b) $k \times 8$ SlidingXOR function

Figure 4.7: SlidinXOR function: Number of evaluations needed to find global optimum for increasing number of blocks

4.5 Discussion

There is one interesting question, how is the number of evaluations distributed between LIMD and Local Optimizer. An answer on this question depends on the LIMD algorithm, because the longer the dependencies between variables are the bigger number of evaluations is consumed by Local Optimizer and the less number of evaluations is consumed by LIMD. If LIMD finds in every iteration new dependency between variables, the distribution curve will be oscillate around middle. But if new dependency isn't found in every iteration, the LIMD algorithm will consume more and more evaluations. On Figure 4.8 is shown how is distributed the number of the evaluations between LIMD and Local Optimizer through solving 4x8bitTrap function in themselves ratio, where ratio is $\frac{NEvals_LO}{NEvals_LIMD}$.



(a) 4x8bitTrap function: LIMD BSF continue version

(b) 4x8bitTrap function: Randomize version

Figure 4.8: Trap function: Distribution of evaluation between LIMD and Local Optimizer. High half: LIMD, bottom half: Local Optimizer

Chapter 5

Conclusion

In this thesis was described an optimization algorithm using local search and linkage learning in a discrete binary world. This algorithm had two parts, a local search algorithm and a linkage learning algorithm. The first part, local search algorithm, works under groups of bits, this groups are created by dependent variables. The LIMD algorithm gives us information which variables are dependent. There was two versions of the algorithm, first version tried to use some kind of heuristic (continue from best so far solution) and second version of the algorithm was completely random.

We have tested this algorithm on several problems against two genetic algorithms. The first group of problems were additively decomposable functions, described in Chapter 2 Section 2.1. The second group of problems were hierarchical problems, described in Chapter 2 Section 2.2.

Subject to testing algorithm was to find out how much evaluations is needed to find global optima in thirty out of thirty-wasting. There was also convergence graphs and graphs showing the distribution of evaluations.

5.1 Results

I would say, an algorithm described in this paper returns reliable results within a reasonable time, in compare to both other algorithms. Particularly well behaved in this algorithm to the problem of traps, with increasing chain blocks had to find a solution even if other algorithms failing contra hierarchical functions. In dealing with the hierarchical functions algorithm failed on chains longer than 8 bits. It is, because of LIMD function, which have big problems with finding dependencies between bits in long chains. When we have a long chain (for example chain with length of 16), there is only a few strings to detect possible dependencies between variables.

People say, that for problems with dependencies between variables are population algorithms the best. But with restarts of the algorithm from random points is possible to substitute some characteristics of the population algorithm and find dependencies between variables. as in this thesis is shown.

It's hard to say, which version of algorithm, random or LIMD best so far continue version, is better on given problem. People would say the longer block of dependent variables the more effective is LIMD best so far continue version, it isn't true. It dependent on the functions character.

This work made me stronger in optimization problems. Also, it gives me experiences how to work on large project and how to describe it. In the end, I think that the goals of the work I was able to meet it with good results, but there is still space for improvement, for example if LIEM or LIEM² was used.

Bibliography

- [1] Thompson, Richard K. and Wright, Alden H., 1997.
Additively Decomposable Fitness Functions, Computer Science Dept., University of Montana.
- [2] Pelikan, Martin, 2005.
Hierarchical Bayesian Optimization Algorithm: Toward a New Generation of Evolutionary Algorithms,
available online: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/3540237747>
- [3] Pošík, Petr, 2010.
Probabilistic models in evolutionary algorithms.
- [4] Pošík, Petr, 2009.
Optimalizace černé skříňky, Přednáška - Aplikace umělé inteligence.
- [5] Watson, Richard A. and Hornby, Gregory S. and Pollack, Jordan B., 1998.
Modeling Building-Block Interdependency, Parallel Problem Solving from Nature ? PPSN V.
- [6] Masaharu Munetomo, David E. Goldberg, 1999.
Identifying Linkage Groups by Nonlinearity/Non-monotonicity Detection, Proceedings of the 1999 Genetic and Evolutionary Computation Conference
- [7] Pelikan, Martin and Goldberg, David E. and Cantù-Paz, Eric, 1998.
Linkage Problem, Distribution Estimation, and Bayesian Networks IlliGAL Report No. 98013, University of Illinois

- [8] Harik, Georges, 1999.
Linkage Learning via Probabilistic Modeling in the ECGA IlliGAL Report No. 99010,
University of Illinois
- [9] Masaharu Munetomo, 2002.
Linkage identification based on epistasis measures to realize efficient genetic algorithms
Congress on Evolutionary Computation 2002
- [10] H. Mühlenbein and G. Paaß, 1996.
From recombination of genes to the estimation of distributions I. binary parameters
Springer-Verlag, Berlin

Appendix A

Content provided CD

This work is accompanied by a CD, which includes the source code and the text of the thesis.

- Folder “Code ”: The source code of the algorithm
- Folder “Data ”: Data for the graphs
- Folder “Text ”: The text of the thesis