Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

Perturbative Linkage Learning for Black-box Optimization of Pseudo-Boolean Functions

BACHELOR THESIS

Author:	Vojtěch Vaněček
Supervisor:	Ing. Petr Pošík, Ph.D.

May, 2023

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Vaněček Vojtěch**	Personal ID number:	**483464**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Perturbative Linkage Learning for Black-box Optimization of Pseudo-Boolean Functions

Bachelor’s thesis title in Czech:

Perturbační detekce závislostí v black-box optimalizaci pseudo-boolovských funkcí

Guidelines:

1. Get acquainted with the Dark-grey Genetic Algorithm (dgGA) for the optimization of pseudo-boolean functions.
2. Survey and analyze perturbative techniques used to detect dependencies among variables.
3. Propose and implement modifications to dgGA (e.g., different method to detect dependencies, different evolutionary model).
4. Evaluate the effects of the modifications on a diverse set of pseudo-boolean functions (Max3SAT, Ising spin glasses, Mk landscapes).

Bibliography / sources:

[1] Przewozniczek, M. et al. On turning Black- Into Dark Gray-optimization with the Direct Empirical Linkage Discovery and Partition Crossover. ACM GECCO 2022
[2] Whitley, D.L. et al. Gray box optimization for Mk Landscapes, Nk Landscapes, and Max-Ksat. Evolutionary Computation 24, MIT Press, 2016.[
[3] Tinos, R. et al. Partition Crossover for Pseudo-boolean Optimization. ACM FOGA, 2015.

Name and workplace of bachelor’s thesis supervisor:

Ing. Petr Pošík, Ph.D. Department of Cybernetics FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **01.02.2023** Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Ing. Petr Pošík, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student’s signature

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 26, 2023 ....
Signature

Acknowledgment

Firstly, I would like to express my gratitude to my supervisor Ing. Petr Pošík, Ph.D. for his feedback and guidance during the writing of this thesis. I am also thankful to my family for supporting me during my studies. Finally, I would like to thank my friends for all the unconditional support in this very intense academic year.

Vojtěch Vaněček

Title:
Perturbative Linkage Learning for Black-box Optimization of Pseudo-Boolean Functions

Author:	Vojtěch Vaněček
Study programme:	Open Informatics
Specialization:	Artificial Intelligence and Computer Science
Publication type:	Bachelor thesis
Supervisor:	Ing. Petr Pošík, Ph.D. Department of Cybernetics
Key words (English):	Linkage learning, genetic algorithms, Gray-box optimization, epistasis measure
Key words (Czech):	Detekce závislostí, genetický algoritmus, Gray-box optimalizace, míra epistáze

Abstract (English): This thesis focuses on the optimization of pseudo-boolean functions using the recently introduced Dark Gray genetic algorithm, which decomposes the problem being solved during the optimization process. This learned, initially covert, underlying structure of the optimized problem then allows the use of Gray-box optimization techniques. To take advantage of these techniques, we describe well-known perturbation-based techniques for detecting interactions among the variables. Subsequently, we describe and analyze the Dark Gray genetic algorithm. The result of this analysis led us to propose a modification in the interaction learning procedure, which we then implemented to the genetic algorithm. Finally, in this thesis we present how the modifications made influenced the optimization performance on various pseudo-boolean functions.

Abstract (Czech): Tato práce se zabývá optimalizací pseudo-boolovských funkcí pomocí, nedávno představeného, genetického algoritmu Dark Gray, který v průběhu optimalizace dekomponuje řešený problém. Tato naučená, původně neznámá, struktura řešeného problému následně umožní využít Gray-box optimalizačních technik. Abychom mohli využít těchto technik, popisujeme známé techniky pro detekci interakcí mezi jednotlivými proměnnými založené na perturbaci. Následně jsme popsali a analyzovali genetický algoritmus Dark Gray. Výsledek této analýzy vedl k návrhu úpravy v postupu učení interakcí, který jsme následně implementovali do genetického algoritmu. Nakonec v práci uvádíme jak provedené úpravy ovlivnili průběh optimalizace různých pseudo-boolovských funkcí.

Contents

Chapter 1

Introduction

Generally in optimization of black-box functions, no underlying structure of the problem is known in advance. This makes the optimization of such a function more difficult, but on the other hand, this general formulation can be used to solve a broader class of problems. In this thesis, we restrict ourselves to the optimization of pseudo-boolean functions. Gray-box optimization [1] covers methods allowing us to optimize pseudo-boolean functions more efficiently, however, in this case, knowledge of the internal structure of the problem is already assumed, so the optimized function can no longer be called a black-box function.

To employ gray-box optimization techniques into black-box optimization, knowledge of the underlying structure of the problem is required. By this underlying structure we mean the direct dependencies among the variables, so that the optimized function is then additively separable. Dark Gray Genetic Algorithm (dgGA) [2] applies gray-box optimization techniques to black-box problems by learning the problem structure during the optimization run. This algorithm is able to detect that a dependency is missing and then correctly identify it. It turns out that this algorithm can compete in many cases with other GA-based optimizers using statistical linkage learning to decompose the problem, such as cGOMEA [3].

In designing a black-box optimizer, we want to reach the optimum after the fewest number of evaluations of the optimized function, since these evaluations are generally computationally expensive. In dgGA, we also monitor the number of function evaluations needed purely to identify interactions between variables. The results presented for dgGA showed that for certain problem classes the number of function evaluations spent on interaction identification was much higher than for other problem classes, which affected the overall efficiency of the optimizer. This observation forms a motivation for our work. **Our goal is to propose modifications to the interaction learning process such that this cost is reduced while not degrading the quality of the problem decomposition.**

The organization of the text is as follows, in Chapter 2 we define the problem of optimization of pseudo-boolean functions and describe the gray-box optimization techniques used in dgGA. In Chapter 3, we describe some well-known perturbation-based techniques for learning interactions among variables. Chapter 4 describes the aforementioned dgGA with its key components. In Chapter 5, we present the results of our analysis on dgGA, which further leads to the introduction of Greedy Linkage Learning into dgGA. This chapter describes our modification in detail. In Chapter 6, we present results comparing the original dgGA and the modified dgGA.

Chapter 2

Pseudo-Boolean Optimization

2.1 Pseudo-Boolean Optimization

2.1.1 Pseudo-Boolean Functions

A pseudo-boolean function is a mapping from a binary string to real values. We consider a binary string to be a tuple of a finite number ($n \in \mathbb{N}$) of binary values $\{0, 1\}$. When we refer directly to binary values in this context, we will usually name them as bits or individual variables of the solution.

$$f : \{0, 1\}^n \rightarrow \mathbb{R} \tag{2.1}$$

There are many research areas where psedo-Boolean functions are used, e.g., the maximum satisfiability problem, the maximum cut problem in graph theory, linear 0-1 programming, and many others.

2.1.2 The k-bounded Pseudo-Boolean Function

One special case of the pseudo-Boolean function defined above is k-bounded function. It will become apparent that some of the naturally inspired problems do indeed satisfy this constraint.

Let us consider a pseudo-boolean function

$$f : \{0, 1\}^n \rightarrow \mathbb{R} \tag{2.2}$$

If the function f is k-bounded than f can be decomposed into m subfunctions. Where each decomposed subfunction f_i depends at most on k variables.

$$f(\boldsymbol{x}) = \sum_{i=1}^{m} f_i(\boldsymbol{x}) \tag{2.3}$$

Knowledge of the correct and complete decomposition of the function f seems to be essential in the whole optimization process and can lead to significant savings during the optimization process.

Examples of k-bounded functions

MAX3SAT

The MAX-3SAT problem is one of the typical representatives of k-bounded pseudo-boolean functions [4]. In this case, the function f can be decomposed into m subfunctions, where m corresponds to the number of clauses. Each of these subfunctions corresponds to a single logical clause, and thus depends on only three variables, which implies that k is 3.

For example, consider the following MAX-3SAT problem. The problem consists of 5 logical clauses and 4 logical variables.

$$\begin{aligned} c_1 &= x_1 \vee \neg x_2 \vee \neg x_3 \\ c_2 &= x_2 \vee x_3 \vee \neg x_4 \\ c_3 &= \neg x_1 \vee x_2 \vee x_3 \\ c_4 &= \neg x_1 \vee \neg x_3 \vee x_4 \\ c_5 &= x_2 \vee \neg x_3 \vee \neg x_4 \end{aligned} \tag{2.4}$$

Then the function f can be decomposed as follows:

$$\begin{gathered} f(\boldsymbol{x}) = \sum_{i=1}^{5} f_i(\boldsymbol{x}) = f_1(x_1, x_2, x_3) + f_2(x_2, x_3, x_4) + f_3(x_1, x_2, x_3) + \\ + f_4(x_1, x_3, x_4) + f_5(x_2, x_3, x_4) \\ \forall i : f_i : \{0,1\}^3 \to \{0,1\} \end{gathered} \tag{2.5}$$

$$\begin{aligned} f_1(x_1, x_2, x_3) &= x_1 \vee \neg x_2 \vee \neg x_3 \\ &\vdots \\ f_5(x_2, x_3, x_4) &= x_2 \vee \neg x_3 \vee \neg x_4 \end{aligned} \tag{2.6}$$

The goal is to find a variable assignment that satisfies as many clauses as possible.

$$\boldsymbol{x}^* = \underset{\boldsymbol{x} \in \{0,1\}^4}{\operatorname{argmax}} f(\boldsymbol{x}) \tag{2.7}$$

It should be noted here that MAX3SAT problems are part of models called Maximum satisfiability, which generalize the NP-complete satisfiability problem.

Mk Landscapes

A new term has been proposed in [1] that generalizes any optimization of the k-bounded pseudo-boolean problem to Mk Landscapes.

$$f(\boldsymbol{x}) = \sum_{i=1}^{M} f_i(\boldsymbol{x}) \tag{2.8}$$

Where M is the number of subfunctions the function f decomposes into, and k is the upper bound on the occurrence of variables in each subfunction f_i. Mk Landscapes generalizes the MAXkSAT problems and also, for example, Ising spin glass problem.

2.1.3 Pseudo-Boolean Optimization

The pseudo-boolean function was originally introduced by Hammer, Rosenberg and Rudeanu in 1963. At the turn of the millennium, Hammer and Boros presented their survey of pseudo-boolean optimization [4], in which they detailed the current state of the art in solving problems related to pseudo-boolean optimization, as well as special cases of this optimization, such as quadratic pseudo-boolean optimization.

The most general formulation of the pseudo-boolean function optimization problem is given as follows:

$$\boldsymbol{x}^* = \operatorname*{argmin}_{\boldsymbol{x}\in\{0,1\}^n} f(\boldsymbol{x}) \qquad (\text{or } \boldsymbol{x}^* = \operatorname*{argmax}_{\boldsymbol{x}\in\{0,1\}^n} f(\boldsymbol{x})) \tag{2.9}$$

In the above formulation, we only consider the unconstrained optimization case, since we will only be interested in problems with unconstrained search space.

2.2 Black-box Optimization of Pseudo-Boolean Functions

In general, the most common approach to optimization is to optimize the "black box" function. We do not place any constraints on the objective function. We can only pass some value arguments to the objective function and get back some number. Since we do not know the internal structure of the function, we call the objective function the "blackbox".

A critical problem in optimizing the black box function can be the very high cost of evaluating the function. By evaluation cost, we mean the computational cost that measures all the resources required to evaluate the function. The most significant advantage of the blackbox optimizer is its generality, but in most cases, more sophisticated optimizers that somehow exploit the internal structure of the problem domain outperform the blackbox optimizer.

Note here that overall black-box optimization is a large area of research with applications almost everywhere. Some of these applications have been described in more detail in [5]. For example, in [6], the authors try to find the best recipe for the tastiest cookie. This problem can also be viewed as a black-box optimization problem. The taste of the resulting cookie can be considered as an objective function that is "evaluated" by the participants who taste the cookie. Which also shows that evaluating the black-box function can be quite costly.

2.3 Gray-box Optimization with Genetic Algorithm

Unlike black box optimizers, we want to take advantage of the structure of the problem and use it in the optimization process. On the other hand, we want to maintain as much generality as possible. This is why we will be interested in the k-bounded pseudo-boolean function, where the design of a graybox optimizer could lead to dramatic savings during the optimization process.

2.3.1 Variables interactions

Under the previous assumption that the optimized function f can be decomposed. We can now define what we consider to be the interaction of variables and how to store these interactions in the data structure, as these interactions will be important in the genetic algorithm darkgray reported here.

For an optimized k-bounded pseudo boolean function, we call two variables interact if they occur in the same subfunction. Furthermore, we will use multiple terms in this thesis depending on the context for almost the same thing. Thus, if two variables interact, we can also say that the first variable is linked to the second, and vice versa, or similarly we can say that one variable is dependent on another. We will also sometimes refer to a building block (BB) as a set of dependent (linked) variables.

Direct and indirect linkage

In addition, when examining these linkages in more depth, we will consider three types of these linkages. The *direct linkage* is the case where both variables occur in the same subfunction. The *indirect linkage* is the case where there is a finite sequence of variable links through which the variables are linked.

So, for example, if it is the case that the variables x_1 and x_2 are directly linked, and likewise x_2 and x_3, then we can say that the variables x_1 and x_3 are indirectly linked.

Add to that, in linkage identification, we will use the term *false linkage* for the situation where two truly independent variables are identified as dependent. The *missing linkage* term for the situation where some linkage remains undetected.

From the example above, when we considered the MAX3SAT problem. The objective function f was decomposed into 5 subfunctions f_1 to f_5. The decomposition of the function f was defined as follows:

$$\begin{aligned} f(\boldsymbol{x}) = f_1(x_1, x_2, x_3) + f_2(x_2, x_3, x_4) + f_3(x_1, x_2, x_3) + \\ + f_4(x_1, x_3, x_4) + f_5(x_2, x_3, x_4) \end{aligned} \tag{2.10}$$

Variables that occur in the same subfunction interact with each other. For example, in subfunction f_4, only variables x_1, x_2, x_3 occur, so they are directly linked with each other.

We will store these types of variable interactions in a variable interaction graph. The variable interaction graph (VIG) is a graph $G = (V, E)$, where V is the set of all variables occurring in the function f and E is the set of just those pairs of variables that are directly linked.

Figure 2.1 shows the Variable Interaction Graph (VIG) for the above MAX3SAT problem (2.10).

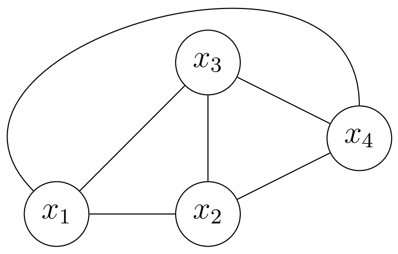


Figure 2.1: Example of a Variable Interaction Graph (VIG) showing the interaction of variables in the above MAX3SAT problem.

This graph can also be represented by an adjacency matrix.

$$adj(G) = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \tag{2.11}$$

The adjacency matrix in 2.11 showing the same graph as in Figure 2.1. The matrix has 1 in the i-th row and j-th column if and only if there is an edge between the i-th and j-th vertices of the graph, or we can say that the i-th and j-th variables are directly linked.

In the next chapter, where we discuss well-known approaches to learning variable interactions, we will see that some of these methods use a similar matrix, but name it the dependency matrix. Other methods have added a measure of the strength of the interaction of the variables, so that the graph then becomes weighted. The edges in it have a weight that represents the strength of the interaction between the connecting variables.

2.3.2 Partition Crossover

The key idea of any genetic algorithm is the process of recombination of the parents' solutions to produce offspring. We use the crossover operator to recombine the parents. Since we consider a gray optimization problem, we assume that the problem is decomposable and there is a linkage between the variables. Then we can use the Partition Crossover (PX) [7] operator to recombine the parents. In this operator, as in many others, bits of parents sharing the same value are inherited by offspring. The Partition Crossover operator determines from which parents the remaining bits will be inherited using the known decomposition of the optimized function.

Here we take advantage of the knowledge of the interaction of the variables, since the optimized function is a k-bounded psedo-boolean. In the next section, it will be shown how to obtain the variable interactions directly during the optimization.

For now, let us assume that we know the correct decomposition of the optimized function and that we have a Variable Interaction Graph (VIG). Furthermore, let us assume that the parents we want to recombine are locally optimal with respect to the neighborhood of Hamming distance 1.

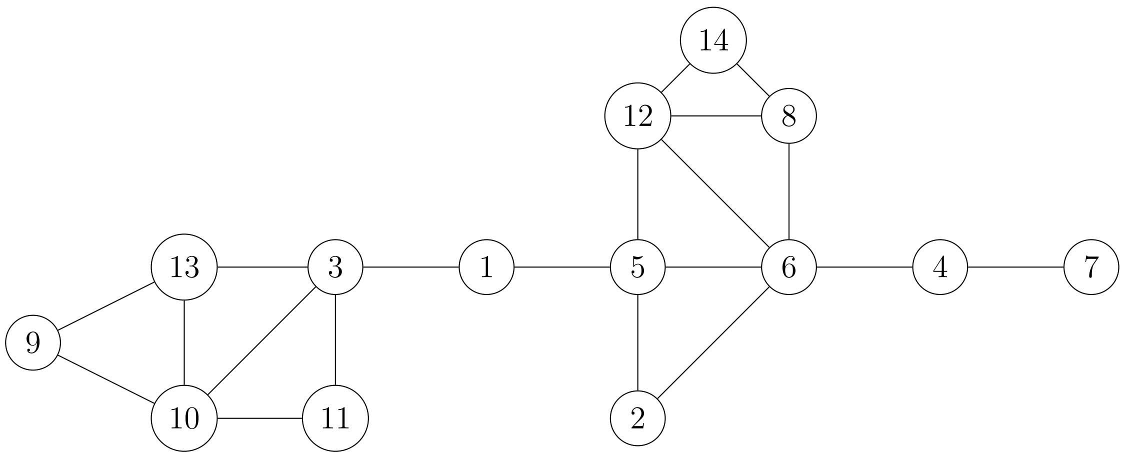
Recombination Graph

Suppose we want to recombine two solutions $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{B}^n$ with the given Variable Interaction Graph G. Then we define a graph called the recombination graph, which is a subgraph of the graph G obtained by removing all vertices representing variables in which the solutions $\boldsymbol{x}$ and $\boldsymbol{y}$ share the same value. All corresponding edges are also removed.

We will show the whole process of generating a recombination graph with an example. Assume two parent solutions $\boldsymbol{x}$ and $\boldsymbol{y}$.

$$\begin{aligned} \boldsymbol{x} &= 01001110010101 \\ \boldsymbol{y} &= 01111100101111 \end{aligned} \tag{2.12}$$

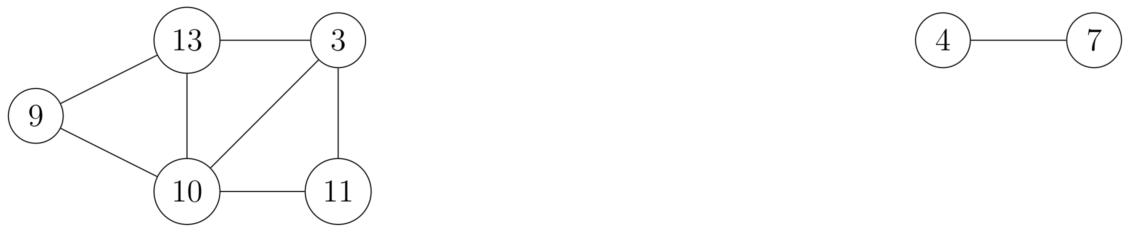
And the following Variable Interaction Graph (VIG)



Now we see that the solutions $\boldsymbol{x}$ and $\boldsymbol{y}$ differ in some bits. Let us illustrate this situation with the solution $\boldsymbol{z}$. The solution $\boldsymbol{z}$ inherits all the bits that have the same values in $\boldsymbol{x}$ and $\boldsymbol{y}$. In the bits where $\boldsymbol{x}$ and $\boldsymbol{y}$ differ, $\boldsymbol{z}$ has * for purposes of illustration.

$$\boldsymbol{z} = 01 * *11 * 0 * * * 1 * 1 \tag{2.13}$$

The variables are numbered from 1 to 14 from left to right. By removing the variables 1, 2, 5, 6, 8, 12 and 14 from the VIG, we obtain the recombination graph. These are the positions where $\boldsymbol{x}$ and $\boldsymbol{y}$ have the same values.



This is the resultant recombination graph. Finally, we should note that the recombination graph has two connected components, namely {3, 9, 10, 11, 13}, {4, 7}.

Recombination

Now we can move on to a general description of the recombination process. Suppose we want to recombine two parent solutions $\boldsymbol{x}$ and $\boldsymbol{y}$ and obtain a child $\boldsymbol{z}$. First, the bits with a common assignment in both parents are inherited directly by $\boldsymbol{z}$. For the other bits, we create a recombination graph from the predefined VIG. Since we know that the optimized function f is decomposable, each connected component of the recombination graph representing the variables of the function f is independent from the other variables. Then, for each component, we can decide whether it is better to inherit bits from solution $\boldsymbol{x}$ or $\boldsymbol{y}$, simply by evaluating the function f with both assignments.

To illustrate, we can continue with the above example. The recombination graph gives three connected components. Thus, we know that variables 3, 9, 10, 11 and 12 interact with each other, as well as variables 4 and 7. Taking the first component of the recombination graph. We want to choose whether it is better to inherit these bits from $\boldsymbol{x}$ or $\boldsymbol{y}$, which are shown below as red "$*$".

$$\boldsymbol{z} = 01{*}*11*0{*}{*}{*}1{*}1 \tag{2.14}$$

Since we know the decomposition of the optimized function f, it is sufficient to compare the value of the function when the bits are inherited from the solution $\boldsymbol{x}$ or $\boldsymbol{y}$. Here we assume maximization of the function f.

$$\boldsymbol{z} = 0100*11*0010101 \Leftrightarrow f_1(x_3, x_9, x_{10}, x_{11}, x_{13}) > f_1(y_3, y_9, y_{10}, y_{11}, y_{13}) \tag{2.15}$$

If the above inequality (2.15) does not hold, the bits are inherited from $\boldsymbol{y}$. Next, we repeat the same procedure for the second component.

$$\boldsymbol{z} = 01*01110***1*1 \Leftrightarrow f_2(x_4, x_7) > f_2(y_4, y_7) \tag{2.16}$$

Now the recombination process is complete and the offspring $\boldsymbol{z}$ solution is formed.

Locally Optimal Offspring

In the previously mentioned paper where Partition Crossover [7] was proposed, it was proven that if we recombine parents that are both locally optimal, the offspring generated will also be locally optimal. When we refer here to a local optimum, we consider it as an individual (solution) that is a local optimum with respect to the Hamming distance 1 neighborhood. Thus, for this individual, flipping any of its bits would not lead to a better value of the objective function. In practice, when the complete VIG is known, we apply Partition Crossover only to parents that are both locally optimal. Thus, we optimize the parents using greedy local search before the recombination process.

Chapter 3

Perturbation-based Linkage Detection Techniques

This chapter surveys the linkage learning techniques mainly based on the perturbation. However it will be shown that there are several techniques that supplies perturbation as the main component of linkage identification with other techniques. The accurancy of the reported methods is mostly based on the population size. By population we mean finite set of the individuals (solutions). We should also percieve and consider two types of the fitness functions, that with overlaps and without overlaps. In this case by overlap we mean that the function decomposition has at least two BBs with nonempty intersection, meaning that there are exists at least two subfunction having same argument variable. For example in the previous chapter we mentined MAX3SAT problem which could be seen as the one of many problems with overlapping blocks.

3.1 Identifying Linkage by Nonlinearity Check

In 1998, Munetomo and Goldberg proposed a simple procedure [8] that identifies linkages among the variables. The presented procedure is called Linkage Identification by Nonlinearity Check (LINC).

3.1.1 Linearity and Nonlinearity

The LINC procedure works under the assumption that a nonlinear interaction exists between at most k bits. (Since we are primarily interested in optimizing the boolean function, we will sometimes refer to the bits of the solution instead of specific decision variables.) These grouped, interacting bits form a so-called building block (BB).

Consider the following solution $\boldsymbol{x}$ of length n and a fitness function f satisfying the above assumptions.

$$\boldsymbol{x} = x_1 x_2 x_3 \cdots x_n \tag{3.1}$$

Now we create a perturbed solution $\boldsymbol{x}_i$ with the i-th bit flipped, $\boldsymbol{x}_j$ with the j-th bit flipped and $\boldsymbol{x}_{i,j}$ with the i-th and j-th bits of the original solution $\boldsymbol{x}$ flipped.

$$\begin{aligned} \boldsymbol{x}_i &= x_1 \cdots \bar{x}_i \cdots x_n \\ \boldsymbol{x}_j &= x_1 \cdots \bar{x}_j \cdots x_n \\ \boldsymbol{x}_{i,j} &= x_1 \cdots \bar{x}_i \cdots \bar{x}_j \cdots x_n \end{aligned} \tag{3.2}$$

Finally, we compute the fitness of these generated perturbed solutions and the original solution, focusing on the fitness growth obtained by perturbation.

$$\begin{aligned} \Delta f_i &= f(\boldsymbol{x}_i) - f(\boldsymbol{x}) \\ \Delta f_j &= f(\boldsymbol{x}_j) - f(\boldsymbol{x}) \\ \Delta f_{i,j} &= f(\boldsymbol{x}_{i,j}) - f(\boldsymbol{x}) \end{aligned} \tag{3.3}$$

If perturbing the i-th and j-th bits causes additive fitness growth, then we speak about linearity. Otherwise, a nonlinear effect is observed.

$$\begin{aligned} \Delta f_{i,j} &= \Delta f_i + \Delta f_j \text{ (linearity)} \\ \Delta f_{i,j} &\neq \Delta f_i + \Delta f_j \text{ (nonlinearity)} \end{aligned} \tag{3.4}$$

We consider the i-th and j-th bits as dependent if their perturbation shows a non-linear effect. If the perturbation causes a linear effect, we cannot say that the i-th and j-th bits are necessarily independent, but the linearity is caused in the current context.

To better understand this problem, let us consider a order-k deceptive trap function with $k = 4$, which is defined as follows.

$$f_{deceptiveTrap}(\boldsymbol{x}) = \begin{cases} k & , u(\boldsymbol{x}) = k \\ k - 1 - u(\boldsymbol{x}) & , u(\boldsymbol{x}) < k \end{cases} \tag{3.5}$$

Where u is a unitation function and is defined as the number of ones in $\boldsymbol{x}$.

Assume the following solution $\boldsymbol{x} = 0010$. We want to find out if there are linkages between the variables. In this case, we check the second and fourth bits of the solution. We will see that in some cases the perturbation of the bits that are dependent from the problem definition causes only a linear effect.

$$\begin{aligned} \Delta f_1 &= f(0,1,1,0) - f(0,0,1,0) = (4-1-2) - (4-1-1) = 1 - 2 = -1 \\ \Delta f_3 &= f(0,0,1,1) - f(0,0,1,0) = (4-1-2) - (4-1-1) = 1 - 2 = -1 \\ \Delta f_{1,3} &= f(0,1,1,1) - f(0,0,1,0) = (4-1-3) - (4-1-1) = 0 - 2 = -2 \\ \Delta f_{1,3} &= -2 = -1 - 1 = \Delta f_1 + \Delta f_3 \end{aligned} \tag{3.6}$$

As we see from above equation the perturbation in the pair of bits caused only linear effect in the overall fitness function but we know that bits are dependent. Thus, in this case, multiple solutions need to be checked to detect the non-linearity effect of the bits perturbation. Specifically in this case, it would be solution $\boldsymbol{x} = 1010$.

Therefore, in order to obtain the linkages between the variables, it is necessary to check the proper sized population. The proper population size is more clearly described in [8]. As well as the size of the allowable detection error for a fitness function with overlapping BBs.

3.1.2 Linkage Identification

In the implementation introduced in [8], three sets are prepared at the beginning, namely *linkage set*, *unlikage set* and *unlinkage set refused.* These sets are initialized for each bit. First, it is checked whether the bit perturbation causes a nonlinear effect for the whole population. These pairs of bits are then added to the *linkage set.* Immediately after detecting the nonlinearity, it is also checked whether the pair has an allowable nonlinearity. This is the case when the following holds.

$$(\Delta f_i > 0 \land \Delta f_j > 0) \land (\Delta f_{i,j} > \Delta f_i \land \Delta f_{i,j} > \Delta f_j) \tag{3.7}$$

If an allowable nonlinearity is detected, this bit pair is added to the *unlinkage set*, otherwise it is added to the *unlinkage set refused.* Finally, after checking the entire population. All the same entries that are already stored in *unlikage set refused* are removed from the *unlikage set.* Then those entries that already exist in the *unlinkage set* are excluded from the *linkage set*. The reason for checking the allowable nonlinearity is that the fitness growth when the bits are perturbed may not be exactly additive in general.

3.2 Identifying Composability using Group Perturbation

In [9], Coffin and Clack presented an extension of the LINC algorithm [8]. They first introduced an optimized version of LINC (oLINC) and then a hierarchical version of oLINC (gLINC, "greedy LINC") with significantly better performance than the original LINC.

3.2.1 Optimized Linkage Identification By Nonlinearity Check

In the original version of LINC, the computation goes roughly as follows. For each individual in the population, the individual's pair of bits is checked for nonlinearity without checking whether these bits has been checked before. For this reason, the authors introduced the so-called "non-repetition rule" in the optimized version of LINC (oLINC).

This approach can be easily explained by just checking to ensure that the *linkage set* already contains the pair of bits before it is checked for nonlinearity. If the pair is already present in the *linkage set*, the nonlinearity test is skipped. So the optimized version of LINC avoids unnecessary repetition of the nonlinearity check. This saves the computational cost of the overall learning procedure.

3.2.2 Greedy Linkage Identification By Nonlinearity Check

The aim of the original LINC procedure is to identify linkages among variables. These linked variables form the BBs. When these BBs are obtained, it is at least easier for the GA to find the optimal solution because the problem is reduced to just mixing these BBs in an optimal way. These BBs can be optimized independently of the other BBs. After running the LINC procedure, we only have the linkage sets for each variable. So when it comes to finding the whole BB, we have to go through the linkages to get the complete BB.

Therefore, the authors introduced a new learning method as a combination of the aforementioned oLINC and a new group perturbation linkage learning operator (gLINC). The idea behind this group perturbation is that instead of checking the nonlinearity in each bit, a group of bits will be perturbed simultaneously and then multiple nonlinearities can be observed at once.

Consider the following order-6 trap function, similar to that in 3.5. We already have information about the sets of variables $\{x_1, x_5\}$ and $\{x_3, x_6\}$ forming BBs. From the problem definition here, we know that the problem cannot be decomposed more than into $\{x_1, x_2, x_3, x_4, x_5, x_6\}$.

$$f_{deceptiveTrap}(\boldsymbol{x}) = \begin{cases} 6 & , u(\boldsymbol{x}) = 6 \\ 5 - u(\boldsymbol{x}) & , u(\boldsymbol{x}) < 6 \end{cases} \tag{3.8}$$

Assume also the solution $\boldsymbol{x} = 010100$. We now proceed similarly to the LINC procedure, but instead of perturbing one bit, we perturb the whole set forming BB.

$$\begin{aligned} \Delta f_{\{1,5\}} &= f(1,1,0,1,1,0) - f(0,1,0,1,0,0) = 1 - 3 = -2 \\ \Delta f_{\{3,6\}} &= f(0,1,1,1,0,1) - f(0,1,0,1,0,0) = 1 - 3 = -2 \\ \Delta f_{\{1,5,3,6\}} &= f(1,1,1,1,1,1) - f(0,1,0,1,0,0) = 6 - 3 = 3 \\ \Delta f_{\{1,5,3,6\}} &= 3 \neq -2 - 2 = \Delta f_{\{1,5\}} + \Delta f_{\{3,6\}} \end{aligned} \tag{3.9}$$

As we can see from the last inequality, the group perturbation caused a nonlinear effect, so now we can say that the variables $\{x_1, x_3, x_5, x_6\}$ form a single BB.

More formally, at the beginning we are given two disjunctive sets of variables I and J. We want to know whether these sets form a single BB.

$$\begin{aligned} I &= \{I_1, I_2, \cdots, I_n\} \\ J &= \{J_1, J_2, \cdots, J_m\} \end{aligned} \tag{3.10}$$

To find out, we compute the following fitness function values. Where a block of bits from I, J and both are perturbed. Then we measure the fitness difference between perturbed and non-perturbed individuals.

$$\begin{aligned} \Delta f_I &= f(\boldsymbol{x}_I) - f(\boldsymbol{x}) \\ \Delta f_J &= f(\boldsymbol{x}_J) - f(\boldsymbol{x}) \\ \Delta f_{I \cup J} &= f(\boldsymbol{x}_{I \cup J}) - f(\boldsymbol{x}) \end{aligned} \tag{3.11}$$

The variables from I and J form a single BB if a nonlinear effect is observed.

$$\Delta f_{I \cup J} \neq \Delta f_I + \Delta f_J \text{ (nonlinearity)} \tag{3.12}$$

Similar to the LIEM approach (3.4), this condition (3.12) can be generalized with the δ parameter.

$$|\Delta f_{I \cup J} - (\Delta f_I + \Delta f_J)| > \delta \tag{3.13}$$

3.3 Identifying Linkage Groups by Non-monotonicity Detection

In [10], Munetomo and Goldberg proposed linkage identification using nonmonotonicity detection (LIMD). Similar to LINC, this procedure uses perturbation between a pair of bits to detect group linkage. Furthermore, in this paper, the authors also proposed a tightness detection procedure that measures the tightness of the found linkages in a certain way. This method removes loosely connected group of linkages.

3.3.1 Non-monotonicity Detection

When we described the LINC procedure, we checked whether the perturbation of the bits causes a nonlinear effect in the overall fitness function. Here we will check whether perturbation causes non-monotonicity, or more precisely, whether perturbation does not cause a monotonic increase or decrease of the fitness value. These conditions are formulated below. The notation here is the same as we used to formulate the LINC procedure.

$$(\Delta f_i > 0 \wedge \Delta f_j > 0) \wedge (\Delta f_{i,j} \leq \Delta f_i \wedge \Delta f_{i,j} \leq \Delta f_j) \tag{3.14}$$

$$(\Delta f_i < 0 \wedge \Delta f_j < 0) \wedge (\Delta f_{i,j} \geq \Delta f_i \wedge \Delta f_{i,j} \geq \Delta f_j) \tag{3.15}$$

Linkage between the variables x_i and x_j is detected if any of the above conditions (3.14 or 3.15) holds for any $\boldsymbol{x}$. In other words, if either of the above conditions holds, a series of perturbations in x_i and x_j violates either the monotone increase ($f(\boldsymbol{x}) < f_i(\boldsymbol{x}) < f_{i,j}(\boldsymbol{x}), f(\boldsymbol{x}) < f_j(\boldsymbol{x}) < f_{i,j}(\boldsymbol{x})$) or decrease ($f(\boldsymbol{x}) > f_i(\boldsymbol{x}) > f_{i,j}(\boldsymbol{x}), f(\boldsymbol{x}) > f_j(\boldsymbol{x}) > f_{i,j}(\boldsymbol{x})$) of the fitness value.

Let's take a closer look at the following example. Consider a order-4 deceptive trap function with unitation function $u(\boldsymbol{x})$ that returns the number of ones in a given binary string $\boldsymbol{x}$.

$$f_{deceptiveTrap}(\boldsymbol{x}) = \begin{cases} 4 & , u(\boldsymbol{x}) = 4 \\ 3 - u(\boldsymbol{x}) & , u(\boldsymbol{x}) < 4 \end{cases} \tag{3.16}$$

Next we consider an individual $\boldsymbol{x} = 0101$, we want to inspect the linkage between x_1 and x_3.

$$\begin{aligned} \Delta f_1 &= f(1,1,0,1) - f(0,1,0,1) = 0 - 1 = -1 \\ \Delta f_3 &= f(0,1,1,1) - f(0,1,0,1) = 0 - 1 = -1 \\ \Delta f_{0,2} &= f(1,1,1,1) - f(0,1,0,1) = 4 - 1 = 3 \end{aligned} \tag{3.17}$$

Since the fitness differences caused by the perturbation satisfy condition 3.15, monotonicity is violated and therefore we consider x_1 and x_3 to be linked. Analogous to LINC, these variables are added to the *linkage set.*

3.3.2 Tightness Detection

In [10], the authors focused on learning the linkages of fitness functions composed by overlapped BBs. For this purpose, they proposed a tightness detection procedure to get rid of those linkages that loosely link variables.

The tightness function is given as follows.

$$tightness(i,j) = \frac{n_1(i,j)}{n_1(i,j) + n_2(i,j)} \tag{3.18}$$

Where the function $n_1(i,j)$ is defined as the number of linkage sets that contain both variables i and j. Note here that the *linkage set* exists for each variable separately. The function $n_2(i,j)$ is the number of linkage sets that contain either variable i or j. The above definition of the tightness function implies that the value of the function will be $0 \leq tightness(i,j) \leq 1$.

To be able to modify the linkage information, we need to set the parameter δ, where δ is $0 \leq \delta \leq 1$. In order to get rid of the linkage overspecification, we remove those linkages that violate the following condition.

$$tightness(i,j) < \delta \tag{3.19}$$

The proper population size for the LIMD procedure as well as the δ parameter setting is described in detail in [10], similar to LINC.

3.4 Linkage Identification Based on Epistasis Measure

In [11], Munetomo proposed linkage identification with epistasis measures (LIEM). This method is based on the previously mentioned LINC and on several studies on epistasis measures [12]. These studies usually focus on the relation between the epistasis measure and fitness function while trying to estimate the difficulty of the problem. In short, epistasis is the amount of influence a decision variable, an individual bit, has on other variables.

To achieve this linkage learning procedure, the authors used a simple pairwise epistasis measure using the previously published LINC procedure. For each pair of variables i and j, the pairwise measure of epistasis is defined as follows.

$$e_{i,j} = \max_{\boldsymbol{x} \in P} \quad |\Delta f_{i,j}(\boldsymbol{x}) - (\Delta f_i(\boldsymbol{x}) + \Delta f_j(\boldsymbol{x}))| \tag{3.20}$$

The notation here is the same as we used for LINC, except that here the deltas, i.e., the differences in the value of the fitness function, are functions for each individual $\boldsymbol{x}$ in population P. We maximize the above expression over the entire population of individuals, which implies that, as for LINC and LIMD, the discovery of linkages between variables depends on the size of the population.

$$\begin{cases} \text{i and j are tightly linked} & \text{, if } e_{i,j} \neq 0 \\ \text{i and j are not linked} & \text{otherwise} \end{cases} \tag{3.21}$$

If the epistasis value for variables i and j is non-zero, i.e. i and j have a non-linear interaction, then i and j are members of a single BB. Otherwise, they should be treated separately because the perturbation of the i-th and j-th bits shows linearity ($\Delta f_{i,j}(\boldsymbol{x}) = \Delta f_i(\boldsymbol{x}) + \Delta f_j(\boldsymbol{x})$).

The advantage of the presented procedure is that we can observe the epistasis measure of variables i and j. If the value is small then the amount of nonlinearity caused by the perturbation is small and we can say that the linkage between i and is small. Thus, we can determine the measure of epistasis from when we no longer consider the variables as dependent, part of a single BB or we can also select a number of linkages having the largest measure of epistasis.

3.5 Linkage Identification by Fitness Difference Clustering

In the [13] paper, authors Tsuji, Munetomo and Akama introduced a new approach to identifying linkages between variables, namely Dependency Detection for Distribution Derived from fitness Differences (D^5). Unlike the methods discussed so far in this work, the authors combined the perturbation learning method and Estimation of distribution algorithms (EDAs).

3.5.1 Estimation of Distribution Algorithms

So far, all the methods mentioned for learning linkages have been based on perturbation followed by some comparison of the fitness function values caused by this perturbation. Methods based on distribution estimation (EDAs) build a probabilistic model based on individuals in the population. This estimation then helps to locate the next space to be searched in the solution space defined by the given problem domain. The probabilistic model is built at runtime during optimization and becomes more accurate with each successive generation. In [13], the authors made two important points when examining EDAs, namely that these methods are able to learn the structure of a given problem and also that they are very efficient if the fitness contributions are distributed uniformly among the variables.

3.5.2 Dependency Detection for Distribution Derived from fitness Differences

As already mentioned D^5 combines methods based on EDAs and those based on perturbation. EDAs methods detect dependencies very well if the fitness contributions are uniformly distributed. The perturbation-based methods can detect even those BBs of variables whose fitness contribution is small.

The D^5 authors divided the linkage learning process into three phases. The first phase consists of calculating the difference in fitness value caused by the perturbation for each individual in the population, in a similar way to the LINC method. The second phase is to classify the individuals into clusters, sub-populations, according to the fitness differences computed in the first phase. The third phase is to estimate the distribution of each sub-population.

Fitness differences are calculated in the same way as in LINC. Now assume that we want to find out which variables depend on the i-th variable. Then we compute the difference in fitness caused by perturbing the i-th bit of each individual in the population as follows:

$$\begin{aligned}\Delta f_i &= f(\boldsymbol{x}) - f(\boldsymbol{x}_i) \\ &= f(x_1 \cdots x_n) - f(x_1 \cdots \bar{x_i} \cdots x_n)\end{aligned} \tag{3.22}$$

Using these calculated differences, the individuals of the population are then divided into clusters $\mathrm{C}_1, \mathrm{C}_2, \cdots$ The clustering algorithm is initialized so that each individual of the population is assigned to its own cluster. The algorithm then repeats the following steps until the clustering is sufficient. First, it calculates the "distance" between all pairs of clusters. The distance here is the absolute value of the difference between the fitness differences caused by the perturbation. For a cluster, this difference is calculated as the average of the difference of all its assigned individuals. After these distances between clusters are computed, just those pairs of clusters that are "closest" to each other are joined. After joining, the aforementioned cluster averages need to be recalculated. These steps are repeated as long as there are clusters that are close to each other and there are enough clusters. These clusters are disjunctive subsets of the population, so we can also call them sub-populations.

The last stage of dependency discovery is to build a set of linkages (BB of variables). For all sub-populations (clusters) created in the previous step, we repeat the following procedure. At the beginning, we initialize the sets of variables W_1 and W_2 as follows:

$$W_1 = \{1, \cdots, i-1, i+1, \cdots, l\} \quad \text{and} \quad W_2 = \{i\} \tag{3.23}$$

After this initialization, we gradually move elements (variables) from the set W_1 to the set W_2 until there are k elements in it. At each step, we add to W_2 the variable j that gives the smallest entropy $E(W_2 \cup \{j\})$, defined as follows:

$$E(W_2) = -\sum_{x=1}^{2|W_2|} p_x \log_2 p_x \tag{3.24}$$

Where p_x is the appearance ratio of each substring x and $2|W_2|$ is the number of all possible substrings defined by W_2. After performing the above procedure for each sub-population p (cluster C_p). We define V_i as the set of variables dependent on variable i as follows:

$$\begin{aligned}p' &= \operatorname*{argmin}_p E_p \\ V_i &:= W_{p'}\end{aligned} \tag{3.25}$$

After the construction of these linkage sets for each variable, the linkages are transferred to the so-called dependency matrix, which is then further used in BBs mixing. This matrix, of dimension $n \times n$, has 1 in the ith row and jth column if $j \in V_i$ and $i \in V_j$, 0 otherwise. If the found linkages over-specify the problem structure, the tightness detection procedure, the same as in LIEM, is used to remove unnecessary linkages.

3.6 Linkage Learning using the Maximum Spanning Tree of the Dependency Graph

In [14], authors Helmi, Pelikan, and Rahmani introduced a new approach to learning linkages between variables using a maximum spanning tree of the dependency graph. The procedure is divided into three parts, the first is the actual construction of the dependency graph, the second is finding the maximum spanning tree of this graph and finally removing false linkages.

The dependency graph is an undirected, weighted graph of $G = (V, E)$. The set of vertices V corresponds to the particular decision variables. The weight of an edge e_{ij} between i and j is a real number corresponding to the strength of the dependence between variable i and j. This value is obtained by observing the fitness difference caused by the pairwise perturbation, similar to the LIEM.

After the dependency graph is constructed, the maximum spanning tree algorithm is run on it. Since we assume that the optimization problem is somehow decomposable, the dependency graph is decomposed into multiple subgraphs, each corresponding to the dependent variables.

The variable dependency graph encompasses two types of linkage, the correct ones and the false ones. False linkages are those linkages having a small value of e_{ij} (the strength of the linkage between variables i and j is small). In order to get rid of these linkages, we need to find a certain level of linkage strength so that we can remove some of the linkages. To find this parameter, the *k-means* clustering technique with $k = 2$ is used. After finding this level, the individual linkages are separated into two clusters, the correct ones and the false ones. The false ones are removed from the subgraphs obtained in the previous step and finally the BBs of the variables corresponding to the single subgraphs are identified.

3.7 Inductive Linkage Identification

In [15], authors Chuang and Chen came up with an interesting procedure for detecting linkages among the variables, in contrast to the previously known procedures, they used decision trees mainly used in machine learning. This algorithm is called inductive linkage learning (ILI), the authors divided it into the following three parts. In the first step, the fitness value differences caused by the perturbation are computed. In the second, a decision tree is constructed using these differences. Finally, this tree is used to determine which variables constitute the BBs (blocks of dependent variables).

The fitness differences are computed in the same way as in the D^5 algorithm. That is, after the initial initialization of the population, a random variable i is selected, we will try to find the other variables that together with i form BB. Then for the whole population and variable i, the following fitness differences are computed.

$$\begin{aligned}\Delta f_i &= f(\boldsymbol{x}) - f(\boldsymbol{x}_i) \\ &= f(x_1 \cdots x_n) - f(x_1 \cdots \bar{x}_i \cdots x_n)\end{aligned} \tag{3.26}$$

Using these computed differences, a decision tree is learned using the ID3 [16] algorithm. In the following section, we give more detail on how this learning works as a very fundamental component of the algorithm.

Once this decision tree is constructed, the dependent variables (BBs) are then obtained by simply traversing the tree from the root and storing all attributes (nodes) that are not leaves. Each attribute represents exactly one variable.

3.7.1 Decision Tree Learning

Before we at least briefly explain the process of creating a decision tree, let us illustrate how such a tree can look like and what the nodes and edges represent. Consider the following individual $\boldsymbol{x}$, representing a solution to some problem of length 10.

$$\boldsymbol{x} = x_1 x_2 \cdots x_{10} \tag{3.27}$$

For example, after learning the decision tree itself, the tree might look like this:

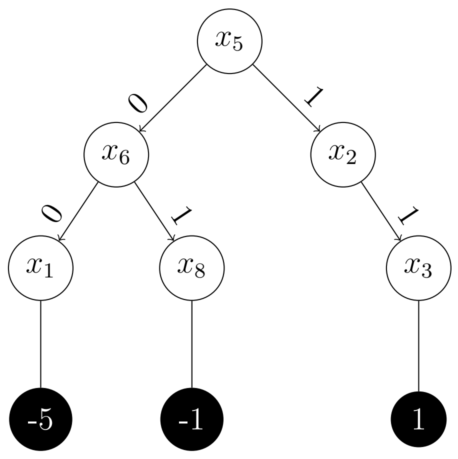


Figure 3.1: Example of a Decision Tree

Each node of the tree represents a decision variable, one bit of the solution. The edges correspond to setting the bit to either one or zero. The leaves of the tree are the predicted fitness values when the solution bits were set adequately. For example, consider individual $\boldsymbol{x}' = 0101010111$, noting that $x_5 = 0$ and $x_6 = 1$. For this individual, the above decision tree predicts that the difference in fitness caused by due to a perturbation of the i-th bit will be -5 and this is based on only two bits of the total number of bits of the solution. The corresponding identified set of linkages will be $\{x_1, x_2, x_3, x_5, x_6, x_8\}$, so this variables forming BB.

As mentioned earlier, ID3 [16] was used for learning the decision tree. This needs a training dataset as it is a supervised learning method. This dataset contains a list of individuals with computed fitness differences caused by perturbations of i-th bit. The tree is then constructed by a top-down approach without backtracking. The basic element of the ID3 algorithm is the selection of attributes, in this context referred to as attribute We understand a single variable (bit) of an individual. Thus, at each node, we look for the best attribute to be selected. We consider the best attribute to be the one that maximizes the *information gain*, which in some way measures the expected reduction in the number of instances. The *information gain* is defined as follows:

$$Gain(D, A) = Entropy(D) - \sum_{v \in Val(A)} \frac{|D_v|}{|D|} Entropy(D_v) \tag{3.28}$$

Where D is the collection of instances (in our case, the list of individuals with computed fitness differences),A is the attribute to be tested. $Val(A)$ is the set of possible values of values of A (in our case only 0 or 1). The entropy of D is given as follows:

$$Entropy(D) = \sum_{i=1}^{c} -p_i \log_2 p_i \tag{3.29}$$

Where c is the number of different fitness difference values in the training data set and p_i is the fraction of instances from D that belong to i. Thus, in this case, p_i is the fraction of individuals from D that have a fitness difference caused by the perturbation equal to ith.

Chapter 4

Dark Gray Genetic Algorithm

4.1 Dark Gray-Box Optimization

In this section, we take a closer look at the Dark Gray Genetic Algorithm (dgGA), which was originally proposed in [2]. The authors of dgGA motivated the design of the dark gray optimizer by using empirical linkage learning techniques proposed in [18, 17]. Since other linkage learning techniques were mostly designed to only approximate the actual linkages between variables, this precluded the use of any gray box mechanism. The above empirical linkage learning technique replaces the approximation with certainty. Thus, it allows the use of Variable Interaction Graph (VIG) and Partition Crossover (PX) for recombination. The VIG is in fact only empirical or estimated, since the true VIG remains hidden, but even an incomplete decomposition of the function seems to be beneficial.

The key concept in solving the optimization of the black-box function is the decomposition problem. Because we know that in black-box function optimization we do not know anything more about the structure of the problem. The proposed mechanism is to transform the black box optimization problem into a gray box problem and then use standard gray box optimization methods such as PX operator. The main idea of the proposed mechanism is to learn the interactions among variables during optimization. More generally, we are trying to decompose an optimization problem that was originally defined as a black-box function. The two key phases of the mechanism are the discovery of linkages and the detection of missing linkages. The detection of missing linkages is part of the recombination process and indicates which pair of variables should be investigated more closely. The Direct Linkage Empirical Discovery (DLED)[17] procedure is executed on the indicated pair of variables. If the discovery procedure confirms the existence of a linkage between the variables, this linkage is added to the VIG.

4.1.1 Overview

At the beginning, when the dgGA optimization algorithm is run, the VIG is initialized without any edges and is therefore just a discrete graph, and this is because we have no knowledge about the structure of the problem. The population is initialized randomly, then optimized by the greedy Local Search (gLS) algorithm. For the sake of better differentiation, we will call this population *MajorPopulation*. The gLS inspects the individual (bit string) and tries to find the optimal setting of each individual bit so that the overall fitness of the individual is maximized or minimized depending on the problem definition. When no improvement step is available, gLS terminates. This procedure is done for each individual in the *MajorPopulation*. Consequently, each individual of the newly initialized *MajorPopulation* is locally optimal with respect to the Hamming distance [1] 1 neighborhood.

When we initialize the dgGA, we also prepare another population, called *BestPopulation*, in which we store the best individuals we have found so far during the

[1] „Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different."[19]

optimization process. In the *BestPopulation*, all individuals have the same fitness value, but they differ from each other.

The optimization process itself can be terminated in several ways. One may be a limit on computational resources, such as computation time or the number of evaluations of the fitness function. Another may be the desired optimal value of the fitness function.

Until the optimization process is completed, the dgGA will repeat the following steps:

- Population mixing of the *MajorPopulation*
- Updates the *BestPopulation* with individuals from the *MajorPopulation*
- Population mixing of the *BestPopulation* creating the *BestPopulationCrossed*
- Updates the *BestPopulation* with individuals from the *BestPopulationCrossed*
- Checks if the *MajorPopulation* is stuck

The population mixing procedure is described in more detail in the following section. Updating the population consists only of keeping the best individuals from the merger of the two populations. We will see in the population mixing procedure that it allows us to indicate that the population is stuck. If this is the case, keeping this population cannot provide a new promising solution. Therefore, we discard this population and reinitialize it, including gLS, doubling the population size.

Population Mixing

The process of population mixing attempts to improve the overall population of individuals. In this process, each individual tries to be improved by other individuals in the population by crossing them using the PX operator to get their offspring. If the improvement succeeds in at least one case, it means that the population is not stuck. Note that we also consider an improvement when the offspring has the same fitness level as the original individual, the parental solution, but differs in its bits. Whenever an individual improves, that offspring replaces the parent in the population.

The improvement of an individual is done with the PX operator. First, all crossover masks are obtained using the PX operator. These masks are then sorted and shuffled. The *Mix and Detect Missing Linkage* process is then performed on all of these masks and the individual, resulting in the actual crossover and the detection of missing linkages in the VIG. Immediately after crossing and obtaining two offspring, the VIG is checked for missing linkage, this detection is described in detail in the following section.

4.2 Missing Linkage Detection

The Missing Linkage Detection [2] mechanism is a key component of the proposed dark-gray genetic algorithm, as we try to turn the black-box problem into a gray-box problem. Our goal is to achieve a correct decomposition of the black-box function. At the beginning of the optimization, we know nothing about the structure underlying the optimized problem. However, we should at least know that the problem is decomposable in some way, so that it is reasonable to use the proposed mechanism.

In the above mentioned paper, this mechanism is part of the population mixing procedure and is used together with the standard PX [7]. Before we start explaining the whole mechanism, we need to introduce a new name for one concept, namely PX mask. This mask is a string of bits and tells us from which parent solution each bit should be inherited by the child. The length of the mask is defined by the length of the parent solution.

Consider the following parent solutions $\boldsymbol{x}$ and $\boldsymbol{y}$.

$$\begin{aligned} \boldsymbol{x} &= 0100011111010110 \\ \boldsymbol{y} &= 0010100100011011 \end{aligned} \tag{4.1}$$

Imagine we want to recombine $\boldsymbol{x}$ and $\boldsymbol{y}$ with the following mask PX_{mask}.

$$PX_{mask} = 0011110111110000 \tag{4.2}$$

To obtain the offspring $\boldsymbol{o}$, we inherit the i-th bit from $\boldsymbol{y}$ if the i-th bit of the PX_{mask} is 1, otherwise we inherit the bit from $\boldsymbol{x}$.

$$\begin{aligned} \boldsymbol{x} &= 0100011111010110 \\ \boldsymbol{y} &= 0010100100011011 \\ \boldsymbol{o} &= 0110101100010110 \end{aligned} \tag{4.3}$$

At the beginning of the missing linkage discovery procedure, we have two parent solutions $\boldsymbol{x}$ and $\boldsymbol{y}$ that we want to recombine using the previously obtained mask PX_{mask} via PX. Using this mask, we can obtain two offspring of solutions $\boldsymbol{o}_1$ and $\boldsymbol{o}_2$ by swapping $\boldsymbol{x}$ and $\boldsymbol{y}$ before recombining. If one of the following conditions holds, we know that we are missing an interaction between variables. So we know that the VIG used by PX is incomplete.

$$f(\boldsymbol{x}) < f(\boldsymbol{o}_1) \wedge f(\boldsymbol{y}) < f(\boldsymbol{o}_2) \qquad or \qquad f(\boldsymbol{x}) > f(\boldsymbol{o}_1) \wedge f(\boldsymbol{y}) > f(\boldsymbol{o}_2) \tag{4.4}$$

If none of the above conditions hold, it means that we either miss no linkage, or we may miss some linkage but the VIG is so good that it may provide a new and better solution. When a missing linkage is detected, the linkage discovery procedure is executed (4.3). The direct linkage discovery procedure is performed only between bits that differ in value in the PX mask. In [2], it is stated that this should not affect learning performance, but can lead to significant savings of computational resources.

Given this, the idea of a missing linkage detection procedure may seem somewhat confusing. Let's at least briefly explain why it works, and clarify the meaning of these conditions (4.4). Assume here $\boldsymbol{x}$ and $\boldsymbol{y}$ of length n.

$$\begin{aligned}\boldsymbol{x} &= x_1 x_2 \dots x_n \\ \boldsymbol{y} &= y_1 y_2 \dots y_n\end{aligned} \tag{4.5}$$

First, let us take a closer look at the first of the above conditions (4.4).

$$(f(\boldsymbol{x}) < f(\boldsymbol{o}_1)) \wedge (f(\boldsymbol{y}) < f(\boldsymbol{o}_2)) \tag{4.6}$$

This condition (4.6) can be rewritten as follows:

$$\begin{aligned}&(f(\boldsymbol{x}) < f((x_1 \wedge y_1) \vee (PX_{mask_1} \wedge y_1), \dots, (x_n \wedge y_n) \vee (PX_{mask_n} \wedge y_n))) \\ &\wedge (f(\boldsymbol{y}) < f((y_1 \wedge x_1) \vee (PX_{mask_1} \wedge x_1), \dots, (y_n \wedge x_n) \vee (PX_{mask_n} \wedge x_n)))\end{aligned} \tag{4.7}$$

Where the i-th variable in the generated offspring $\boldsymbol{o}_1$ is either $(x_i \wedge y_i)$ when x_i and y_i have the same value, or y_i if the i-th variable is part of the exchanging BB, i.e. PX_{mask_i} is 1. Similarly for $\boldsymbol{o}_2$.

The PX mask represents a partial decomposition of the fitness function f. It decomposes f into a set of dependent variables - the exchange BB - and other variables. Thus, the evaluation of f can be done as follows:

$$f(\boldsymbol{x}) = f_{I \setminus I_{PX_{mask}}}(x_k, \dots, x_l) + f_{I_{PX_{mask}}}(x_i, \dots, x_j) \tag{4.8}$$

Where I denotes the set of all variables and $I_{PX_{mask}}$ those variables that are part of PX_{mask}. In the above equation, each of the functions in the subscript has a set of variables on which it depends.

After this formulation, we can return to the condition from 4.7 and rewrite it as follows:

$$\begin{aligned}&(f(\boldsymbol{x}) < \underbrace{f_{I \setminus I_{PX_{mask}}}(x_k, \dots, x_l) + f_{I_{PX_{mask}}}(y_i, \dots, y_j))}_{f(\boldsymbol{o_1})} \\ &\wedge (f(\boldsymbol{y}) < \underbrace{f_{I \setminus I_{PX_{mask}}}(y_k, \dots, y_l) + f_{I_{PX_{mask}}}(x_i, \dots, x_j))}_{f(\boldsymbol{o_2})}\end{aligned} \tag{4.9}$$

So by rewriting the left sides of the inequalities we get the following:

$$\begin{aligned}&(f_{I \setminus I_{PX_{mask}}}(x_k, \dots, x_l) + f_{I_{PX_{mask}}}(x_i, \dots, x_j) \\ &\qquad < f_{I \setminus I_{PX_{mask}}}(x_k, \dots, x_l) + f_{I_{PX_{mask}}}(y_i, \dots, y_j)) \\ &\wedge (f_{I \setminus I_{PX_{mask}}}(y_k, \dots, y_l) + f_{I_{PX_{mask}}}(y_i, \dots, y_j) \\ &\qquad < f_{I \setminus I_{PX_{mask}}}(y_k, \dots, y_l) + f_{I_{PX_{mask}}}(x_i, \dots, x_j))\end{aligned} \tag{4.10}$$

$$\begin{aligned}&(f_{I_{PX_{mask}}}(x_i, \dots, x_j) < f_{I_{PX_{mask}}}(y_i, \dots, y_j)) \\ &\wedge (f_{I_{PX_{mask}}}(y_i, \dots, y_j) < f_{I_{PX_{mask}}}(x_i, \dots, x_j))\end{aligned} \tag{4.11}$$

As we can see, this condition cannot hold simultaneously, so they only hold if there is a missing variable from $I_{PX_{mask}}$.

In words, we can say that the exchange of BB in both individuals $\boldsymbol{x}$ and $\boldsymbol{y}$ leads to a better fitness value of the whole individual. However, this is incorrect because there can only be one improvement in either $\boldsymbol{x}$ or $\boldsymbol{y}$, so it shows that the fitness of an individual cannot be separated into the fitness of BB and the fitness of the other bits. In other words, it shows that the fitness of the variables outside the BB depends on the settings of the variables that form the BB. So now we can conclude that we are missing some linkage between the variables outside the BB and the variables forming the BB.

The second condition of missing linkage detection (4.4) says essentially the same thing. The exchange of BB between individuals $\boldsymbol{x}$ and $\boldsymbol{y}$ led to a deterioration of fitness in both cases. For the same reasons as in the first condition, this cannot happen together. Thus, some linkage is missing.

4.2.1 Example: Overlapping Deceptive Trap Function

Problem Definition

Let us consider a deceptive trap function of order k, where k = 4, which is defined in the following way:

$$f_{deceptiveTrap}(\boldsymbol{x}) = \begin{cases} 4 & , u(\boldsymbol{x}) = 4 \\ 3 - u(\boldsymbol{x}) & , u(\boldsymbol{x}) < 4 \end{cases} \tag{4.12}$$

Where u is a unitation function and is defined as the number of ones in $\boldsymbol{x}$.

Further, we will consider the concatenation of these three functions. For simplicity, these functions overlap by only one bit, and this overlapping bit is the boundary bit. Thus, our fitness function f is defined as follows:

$$f(\boldsymbol{x}) = f_1(x_1, x_2, x_3, x_4) + f_2(x_4, x_5, x_6, x_7) + f_3(x_7, x_8, x_9, x_{10}) \tag{4.13}$$

Where f_1, f_2, f_3 are the deceptive trap functions (4.12). The overlapping variables are those colored in red and blue.

The optimization problem is to find the optimal argument $\boldsymbol{x'}$such that function f is maximized.

$$\boldsymbol{x}^* = \underset{\boldsymbol{x} \in \{0,1\}^{10}}{\operatorname{argmax}} f(\boldsymbol{x}) \tag{4.14}$$

Complete VIG

Now let's see what a complete VIG for this problem would look like.

Now assume that the following individuals $\boldsymbol{x}$ and $\boldsymbol{y}$ recombine immediately after initialization, so that the VIG is empty. Let $\boldsymbol{z}$ be the hyperplane of possible offspring.

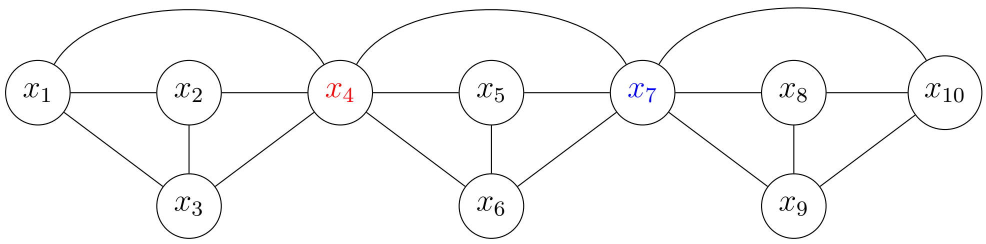


Figure 4.1: The complete VIG showing the interactions among the variables in the problem defined above (4.13).

$$\begin{aligned} \boldsymbol{x} &= 1111001010 \\ \boldsymbol{y} &= 1000101111 \\ \boldsymbol{z} &= 1{*}{*}{*}{*}01{*}1{*} \end{aligned} \tag{4.15}$$

In this case, since the VIG cannot support the decomposition with any knowledge, the components of the recombination graph will be these single-element sets: $\{x_2\}$, $\{x_3\}$, $\{x_4\}$, $\{x_5\}$, $\{x_8\}$ and $\{x_{10}\}$. The PX masks should look like this:

$$\begin{aligned} PX_{mask_1} &= 0000000001 \\ PX_{mask_2} &= 0000000100 \\ PX_{mask_3} &= 0000100000 \\ PX_{mask_4} &= 0001000000 \\ PX_{mask_5} &= 0010000000 \\ PX_{mask_6} &= 0100000000 \end{aligned} \tag{4.16}$$

Now, for example, we use PX_{mask_4} to recombine, so we get two offspring $\boldsymbol{o_1}$ and $\boldsymbol{o_2}$.

$$\begin{aligned} \boldsymbol{o_1} &= 1110001010 \\ \boldsymbol{o_2} &= 1001101111 \end{aligned} \tag{4.17}$$

Finally, we compute the following fitness values to verify the missing linkage condition.

$$\begin{aligned} f(\boldsymbol{x}) &= f_1(1,1,1,1) + f_2(1,0,0,1) + f_3(1,0,1,0) = 4+1+1 = 6 \\ f(\boldsymbol{y}) &= f_1(1,0,0,0) + f_2(0,1,0,1) + f_3(1,1,1,1) = 2+1+4 = 7 \\ f(\boldsymbol{o_1}) &= f_1(1,1,1,0) + f_2(0,0,0,1) + f_3(1,0,1,0) = 0+2+1 = 3 \\ f(\boldsymbol{o_2}) &= f_1(1,0,0,1) + f_2(1,1,0,1) + f_3(1,1,1,1) = 1+0+4 = 5 \end{aligned} \tag{4.18}$$

Because it stands that $f(\boldsymbol{x}) > f(\boldsymbol{o}_1) \wedge f(\boldsymbol{y}) > f(\boldsymbol{o}_2)$ then we may conclude that some linkage missing from the VIG. So with this result, using DLED will be checked if there are any direct linkage between x_4 and $\{x_2, x_3, x_5, x_8, x_{10}\}$.

Now consider the following incomplete but non-empty VIG, which is nothing more than a subgraph of the complete VIG.

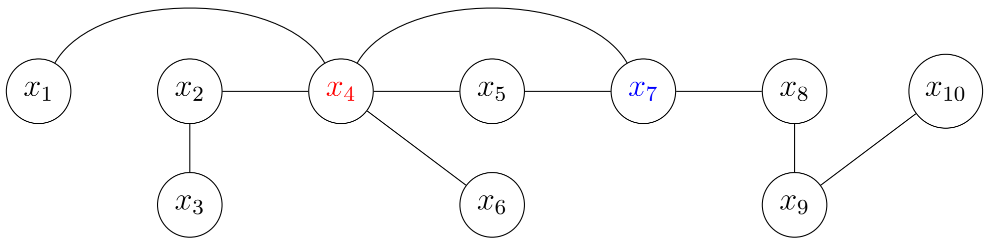


Figure 4.2: Incomplete VIG showing the interactions between variables in the problem defined above.

Assume recombination of the following individuals $\boldsymbol{x}$ and $\boldsymbol{y}$.

$$\begin{aligned}\boldsymbol{x} &= 1010110101\\ \boldsymbol{y} &= 1111010100\\ \boldsymbol{z} &= 1{*}1{*}{*}1010{*}\end{aligned} \tag{4.19}$$

The recombination graph has two components, $\{x_2, x_4, x_5\}$ and $\{x_{10}\}$. PX produces the following masks:

$$\begin{aligned}PX_{mask_1} &= 0101100000\\ PX_{mask_2} &= 0000000001\end{aligned} \tag{4.20}$$

For recombination, we choose PX_{mask_1}, then we get the following descendants.

$$\begin{aligned}\boldsymbol{o_1} &= 1111010101\\ \boldsymbol{o_2} &= 1010110100\end{aligned} \tag{4.21}$$

The values of the fitness function are as follows:

$$f(\boldsymbol{x}) = 3 \quad f(\boldsymbol{y}) = 7 \quad f(\boldsymbol{o_1}) = 6 \quad f(\boldsymbol{o_2}) = 4 \tag{4.22}$$

In this case, neither of the two conditions holds. Therefore, the procedure does not detect the missing linkage.

4.3 Direct Linkage Empirical Discovery

Let us take a closer look at the mechanism for detecting linkages proposed in [17], which was then used in dgGA. For clarity, in this section we will sometimes refer to individual bits of the solution instead of variables. In the above paper, these variables are called solution genes.

Similar to the linkage learning methods described in the previous section, DLED uses bit perturbations. In contrast to those methods already described, DLED tries to discover only *direct linkage* and also often these methods have been introduced only for additive separable problems. However, DLED can also handle the overlapping ones. Since *indirect linkage* would not help us at all, since we try to construct the empirical VIG as accurately as possible during optimization.

The DLED method tests bit against bit to determine the linkage. Thus, to check the linkage between the i-th bit and the j-th bit, we prepare the following perturbed individuals:

$$\begin{aligned} \boldsymbol{x}_i &= x_1 \cdots \bar{x_i} \cdots x_n \\ \boldsymbol{x}_j &= x_1 \cdots \bar{x_j} \cdots x_n \\ \boldsymbol{x}_{i,j} &= x_1 \cdots \bar{x_i} \cdots \bar{x_j} \cdots x_n \end{aligned} \tag{4.23}$$

DLED detects *direct linkage* between variables i and j if the following condition (4.24) hold:

$$(f(\boldsymbol{x}) < f(\boldsymbol{x}_j) \wedge f(\boldsymbol{x}_i) \geq f(\boldsymbol{x}_{i,j})) \vee (f(\boldsymbol{x}) \geq f(\boldsymbol{x}_j) \wedge f(\boldsymbol{x}_i) < f(\boldsymbol{x}_{i,j})) \tag{4.24}$$

The newly found interaction between the i-th and j-th variable can then be stored in the VIG.

Finally, we should mention some properties of the proposed linkage learning mechanism. The DLED procedure detects only *direct linkage* and never reports *false linkage*. This means that the variables identified as dependent actually belong to the same subfunction of the decomposed optimized function. This was proved in the paper that proposed this method. The DLED may miss some linkages, the detection of linkages is dependent on the particular individual solution. Previously used direct linkage learning methods [18] use a local optimizer as a building block. However, the latter has one drawback, it is not guaranteed how many times the optimized function will be evaluated. In contrast, the above approach uses only the exact number of function evaluations. As for 3LO [18], it has been shown that DLED [17] never reports *false linkage*.

In the following section, we will show how the DLED procedure works with an example.

4.3.1 Example: Ising Spin Glass Problem

Problem Definition

Let us now define an optimization problem that is often used as a benchmark for many GA algorithms, for example in [20]. For the purpose of these Ising Spin Glass (ISG) problems, we will need the following mapping:

$$\begin{aligned} T_{ISG} : \{0,1\} &\longrightarrow \{-1,+1\} \\ 0 &\longmapsto -1 \\ 1 &\longmapsto 1 \end{aligned} \tag{4.25}$$

Since in an ISG problem, the solution subspace is not a binary string, but $\{-1,+1\}$ string. The solution subspace is therefore $\{-1,+1\}^n$. From now on, for the sake of clarity, in this example we consider the above mapping in each function evaluation. Because the string $\{-1,+1\}^n$ instead of $\{0,1\}^n$ would look really cluttered.

A graph G is given, which we restrict to a 2D grid. The edges of this graph represent interactions between connected vertices, these vertices are called spins here.

For an instance of the ISG problem with 36 variables, the graph of G might look like this:

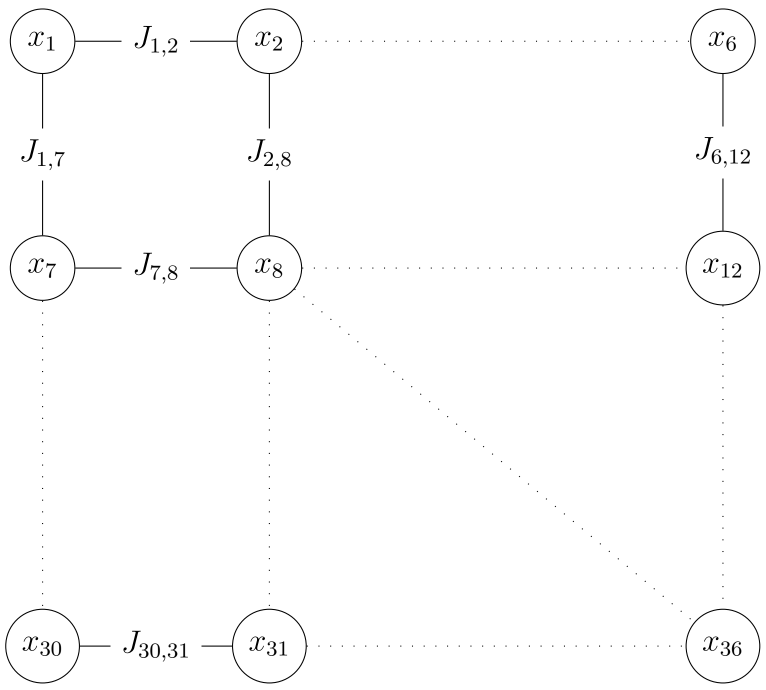


Figure 4.3: Graph showing the interactions between variables in the instance of ISG problem with length 36. $J_{i,j}$ is the coupling constant between x_i and x_j.

The optimization problem related to the ISG problem is defined as follows:

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}\in\{-1,+1\}^n}{\operatorname{argmin}} f(\boldsymbol{x}) \tag{4.26}$$

With the fitness function f defined as follows:

$$f(\boldsymbol{x}) = -\sum_{i,j=1}^{n} x_i x_j J_{i,j} \tag{4.27}$$

$J_{i,j}$ is the coupling constant between the i-th spin (x_i) and the j-th spin (x_j), $n \in \mathbb{N}$ is the size of the problem respectively the number of decision variables (or the length of individual $\boldsymbol{x}$).

Linkage Detection

Let us now consider the exact instance of ISG, for the sake of simplicity with 9 variables and the coupling constants $J_{i,j}$.

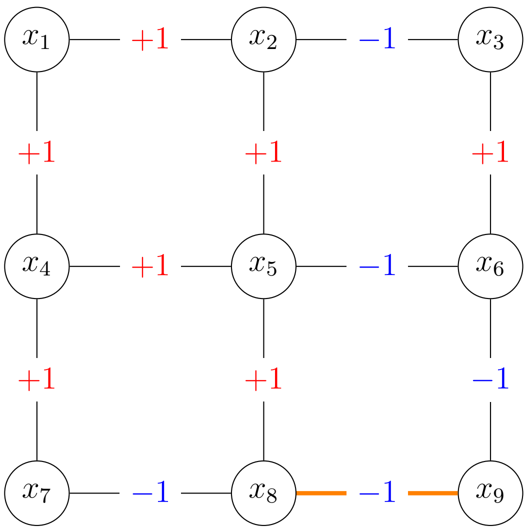


Figure 4.4: For the defined example ISG problem, the graph shows the coupling constants of the respective spins.

From the above definition of the ISG fitness function, it is not hard to deduce that f is decomposable. In this example, f can be decomposed into 12 subfunctions, each of which depends on two variables.

$$\begin{aligned} f(\boldsymbol{x}) = -\sum_{i,j=1}^{n} x_i x_j J_{i,j} = \\ &= -f_1(x_1,x_2) - f_2(x_1,x_3) - f_3(x_2,x_3) - \\ &- f_4(x_2,x_5) - f_5(x_3,x_6) - f_6(x_4,x_7) - \\ &- f_7(x_4,x_5) - f_8(x_5,x_8) - f_9(x_5,x_6) - \\ &- f_{10}(x_7,x_8) - \boldsymbol{f_{11}(x_8,x_9)} - f_{12}(x_6,x_9) \end{aligned} \tag{4.28}$$

Now consider the following individual $\boldsymbol{x}$, we want to check the linkage between x_8 and x_9.

$$\begin{aligned} \boldsymbol{x} &= x_1, x_2, \dots, x_9 \\ \boldsymbol{x} &= 001001010 \end{aligned} \tag{4.29}$$

For this individual $\boldsymbol{x}$, we construct the following individuals with the bits x_8, x_9 and both perturbed.

$$\begin{aligned} \boldsymbol{x}_8 &= 001001000 \\ \boldsymbol{x}_9 &= 001001011 \\ \boldsymbol{x}_{8,9} &= 001001001 \end{aligned} \tag{4.30}$$

Next, we compute their fitness values of these perturbed solutions.

$$\begin{aligned} f(\boldsymbol{x}) &= -10 \\ f(\boldsymbol{x}_8) &= -8 \\ f(\boldsymbol{x}_9) &= -6 \\ f(\boldsymbol{x}_{8,9}) &= -8 \end{aligned} \tag{4.31}$$

Let us recall the first disjunct of the condition for identifying a linkage using DLED (4.24) and consider that $j = 8$ and $i = 9$.

$$(f(\boldsymbol{x}) < f(\boldsymbol{x}_j) \wedge f(\boldsymbol{x}_i) \geq f(\boldsymbol{x}_{i,j})) \tag{4.32}$$

Which is true in this case, so we can conclude that DLED claims that x_8 and x_9 are truly dependent, and this is also consistent with the decomposition we did above (4.28).

Finally, it should be noted that the dependency detection depends on the individual $\boldsymbol{x}$ used.

$$\boldsymbol{x} = 000000000 \tag{4.33}$$

For example, if we run DLED for each variable x_i and x_j, we only get the following dependencies shown in the following VIG.

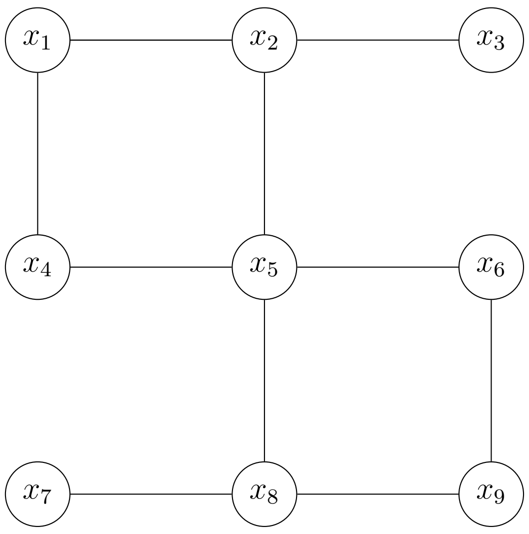


Figure 4.5: The VIG with all detected linkages by the DLED procedure and individual $\boldsymbol{x} = 000000000$.

As we can see, the linkage between x_4 and x_7 is missing, as well as the linkage between x_3 and x_6.

Chapter 5

Dark Gray Genetic Algorithm with Greedy Linkage Identification By Non-linearity Check

In this section, we present the proposed modification to the dgGA. The organization is as follows. We first present some results of the analysis we performed on the baseline dgGA algorithm, based on which we propose improvements that we believe could lead to better performance of the optimizer for some representatives of pseudo-boolean functions. Furthermore, we introduced the greedy linkage learning technique into the dgGA. Finally, we summarize some key features of the proposed modifications to the dgGA.

5.1 Analysis

Our first steps in the analysis led to a revision of the linkage learning procedure. As can be inferred from the results presented in [2], where dgGA was proposed, the number of fitness function evaluations (FFEs) spent on linkage learning itself tends to be quite low for most of the considered problem instances, but for ISG and some cases of concatenation of either deceptive or bimodal functions it still seems to be quite high, around 30% of the total number of FFEs spent during optimization. Thus, we believe that reducing this ratio could lead to an overall better performance of the optimizer.

Since dgGA uses PX as a recombination operator that uses the interactions among the variables stored in the VIG, we only need to use such a linkage learning technique that correctly identifies only the direct linkages among the variables. The linkage learning methods presented in the second chapter mainly identify that the dependent variables are only members of the same BB, which corresponds to a single connected component of the VIG. However, the actual structure of these linkages remains hidden. We also want to avoid detecting false linkage. Thus, replacing DLED with any other method of the linkage learning would not help much.

In dgGA, as mentioned in the previous section, in the population mixing procedure, we try to improve each individual of the population with the other individuals of the population. For each individual, we collect all the PX masks, then shuffle the masks and sort them in ascending order of their length, so we prefer shorter masks, i.e., those that exchange fewer bits. Recall that each of these masks contains, in addition to the recombination mask itself, information identifying the other individual from the population based on which it was generated. We attempt to improve the individual by crossing through these masks and corresponding another individual. Once the crossing over reaches an improvement, the procedure stops. Therefore, the optimal order of these masks is important because we want to achieve improvement as quickly as possible. Based on these considerations, we tried to perform an experiment and estimate the quality of each mask based on the amount of epistasis contained in the linkages and then sort these masks according to their respective qualities. We computed this quality of mask as the sum of the epistasis measures of each pair of variables it contains and then normalized it with its length. To be able to do this we additionally to the already employed VIG, we added another graph storing the epistasis measure between individual variables, computed in same way as LIEM (3.4) does. Higher epistasis measure between variables shows high dependency level between variables. So we thought that inheritance of pair of variables with higher epistasis measure together should be more promising in yielding improved offspring. While this experiment resulted in a better time to reach improvement, fewer masks were tried before finding an improvement, consequently the diversity of the population was reduced and the optimizer tend to get stuck in a local optimum.

We also attempted to speed up linkage learning by implementing DLED directly into the local optimizer used for population initialization. (Recall that the original dgGA only triggers linkage learning when a missing linkage is detected.) In this local optimizer, we cache the fitness values caused by the perturbation until we find a perturbation that improves the fitness of an individual, so that the actual linkage learning consumes only one additional FFE to test a single linkage. Although this learning is less computationally expensive than learning with missing linkage detection, we were unable to accurately estimate when this exploration of new linkages should stop. Nor did it work quite well for the problems with "sparse" VIGs such as ISG. However, we should at least mention that for some problems this leads to better overall optimizer performance, even if the number of FFE spent on linkage learning increases. Moreover, for these problems, it showed that faster discovery of the complete VIG is essential.

Based on these analyses and experiments, we assessed that the most promising next approach to improve dgGA would be to focus on the procedure of learning interactions among variables, improving the efficiency of this learning, i.e., the ratio between the FFEs consumed directly for learning interactions and the total number of FFEs before finding the optimum, without degrading the level of decomposition. Which could then also result in a reduction of the total number of FFEs, hence the overall efficiency of the optimizer.

5.2 Greedy Linkage Learning

Before we go into a detailed description of our modification in dgGA, let us give a few more details about how the linkage learning procedure worked in the original version of dgGA. Below we see the pseudocode of the original implementation of the Mixing and Missing Linkage Discovery procedure.

First, we cross $\boldsymbol{x_1}$, the individual to be improved, and $\boldsymbol{x_2}$, another individual from the population, using *pxMask* to obtain the two offspring $\boldsymbol{x_1'}$ and $\boldsymbol{x_2'}$ in the same way as mentioned in 4.1. (lines 2 and 3) Note that we are only exchanging the bits that are present in the *pxMask*. If this crossover succeeds in improving the fitness value of either parent $\boldsymbol{x_1}$ or $\boldsymbol{x_2}$ then the procedure ends and the improved offspring is returned together with the number that indicates the result of the crossover (line 13 or 17). If the crossover results in a fitness deterioration in both cases, then the conditions for a missing linkage between variables are met (4.4), there is at least one bit missing from *pxMask*. (lines 6 to 11) So if a linkage is missing then we test with DLED if there is a direct linkage between the bit outside the *pxMask* and the bit present in the *pxMask*.(lines 8 to 11) Recall here that we only consider bits outside the *pxMask* to be bits that are part of a crossover, i.e. where $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ differ in

Pseudocode 1 Mixing and Missing Linkage Discovery Procedure

```
1: function MixAndDetectMissingLinks($\boldsymbol{x_1}$, $\boldsymbol{x_2}$, $pxMask$)
2:     $\boldsymbol{x_1^{'}} \leftarrow$ CrossByMask($\boldsymbol{x_1}$, $\boldsymbol{x_2}$, $pxMask$);
3:     $\boldsymbol{x_2^{'}} \leftarrow$ CrossByMask($\boldsymbol{x_2}$, $\boldsymbol{x_1}$, $pxMask$);
4:     if $f(\boldsymbol{x_1}) > f(\boldsymbol{x_1^{'}})$ then
5:         if $f(\boldsymbol{x_2}) > f(\boldsymbol{x_2^{'}})$ then
6:             for $geneM \in pxMask$ do
7:                 for $geneO \notin pxMask$ do
8:                     if $\neg$Dep($empVIG$, $geneM$, $geneO$) then
9:                         DLED($\boldsymbol{x_1}$, $geneM$, $geneO$);
10:                        if $\neg$Dep($empVIG$, $geneM$, $geneO$) then
11:                            DLED($\boldsymbol{x_2}$, $geneM$, $geneO$);
12:        else
13:            return $2, \boldsymbol{x_2^{'}}$;
14:    else
15:        if $f(\boldsymbol{x_1^{'}}) = f(\boldsymbol{x_1})$ then
16:            return $0, \boldsymbol{x_1^{'}}$;
17:        return $1, \boldsymbol{x_1^{'}}$;
18:    return $-1, \boldsymbol{x_1}$;
```

value. If the result of the crossover is the same fitness value, the modified offspring $\boldsymbol{x_1^{'}}$ is returned (line 16). In any other case, the original individual $\boldsymbol{x_1}$ is returned. (line 18)

5.2.1 Our Approach

As we saw above, in the original implementation, if a missing linkage is detected, then it is tested to verify if there is a linkage between the bits in the *pxMask* and the bits outside the *pxMask*. These bits outside the *pxMask* are then tested pairwise with the bits in the *pxMask* so that a direct linkage is detected. However, we find this testing inefficient because, for example, for ISG problems where each bit is dependent on at most four other bits, it is unnecessary to test all bits. Thus, some bits outside of *pxMask* need not be tested at all. More generally, if we assume that the problem to be optimized is somehow decomposable into smaller problems, then we also assume that k (2.1.2) should be at least smaller than the size of the problem to be solved.

For this reason, we have introduced a modified gLINC [9] technique, already described in 3.2.2, into the procedure for discovery of linkages among variables. The idea of the modified gLINC is the same as in the original gLINC, we want to identify whether a given set of bits (variables) form a single BB - i.e., a connected VIG component. However, for our purposes, the condition for identification of the linkage was too strict, so we modified this condition to identify the linkage based on the epistasis measure, similar to the LIEM [11]. That is, if the epistasis measure caused by the perturbation is high then we consider a given set of bits to be a single BB. Using this method, we sequentially divided the bits outside the *pxMask* into those bits that are worth testing, the bits directly dependent on a bit in the *pxMask*, and the remaining bits that do not need to be tested. So we actually use gLINC to filter the bits outside of *pxMask* to only those that really depend on one of the bits in *pxMask*. These bits, which together with the bits in the *pxMask* form the BB, are then further tested using DLED to determine the direct linkage to a particular bit of the *pxMask*.

At the beginning, whenever we detect a missing linkage, we have two disjoint subsets of bits (variables). Let's denote the first as I_{inMask} representing the bits present in *pxMask* and the second subset as $I_{outMask}$ denoting the bits to be tested to see if they are directly linked to any bit from P_{inMask}. In computer science, the following approach is usually known as the Divide-and-conquer algorithm [21]. Once we have both of these sets, we split the set $I_{outMask}$ into two smaller subsets $I_{outMask_1}$ and $I_{outMask_2}$ of the same size. After that, using gLINC, we test whether $I_{outMask_1}$ and I_{inMask} form a single BB. If they form a BB, we continue with this subset $I_{outMask_1}$, which we again recursively split into two smaller subsets. If they do not form a BB together, this subset $I_{outMask_1}$ is discarded. The same steps are applied to the subset $I_{outMask_2}$. These steps are repeated recursively until the subsets form only a single-element set. Finally, we collect these one-element sets. Let us denote I_{dep} as the set of union of these one-element sets. Due to the gLINC technique used, we already know that each bit of the set I_{dep} is directly dependent on at least one of the bits from I_{inMask}, but we do not know which one in specific. Therefore, we need to use DLED to test among which bits from I_{dep} and I_{inMask} there is a direct linkage and update the VIG accordingly.

For the sake of clarity, we have divided the detailed description of the implementation into three parts. The first is the one in which we obtain the promising bits that will be tested for direct linkage with the bit from *pxMask*. The second is the actual implementation of pre-filtering of potentially dependent bits, which excludes less tightly dependent bits from further examination. The third one is responsible for the identification of the BB.

5.2.2 Mixing and Missing Linkage Discovery Procedure with gLINC

Pseudocode 2 Mixing and Missing Linkage Discovery Procedure with gLINC

```
function MixAndDetectMissingLinks($\boldsymbol{x_1}$, $\boldsymbol{x_2}$, $pxMask$)
    $\boldsymbol{x_1'}$ ← CrossByMask($\boldsymbol{x_1}$, $\boldsymbol{x_2}$, $pxMask$);
    $\boldsymbol{x_2'}$ ← CrossByMask($\boldsymbol{x_2}$, $\boldsymbol{x_1}$, $pxMask$);
    if $f(\boldsymbol{x_1}) > f(\boldsymbol{x_1'})$ then
        if $f(\boldsymbol{x_2}) > f(\boldsymbol{x_2'})$ then
            $bitsToBeTested$ ← getBitsToBeTested($\boldsymbol{x_1}$, $\boldsymbol{x_2}$, $pxMask$);
            $bitsFormingBB$ ← GetBitsFormBB($\boldsymbol{x_1}$, $\boldsymbol{x_2}$, $bitsToBeTested$, $pxMask$);
            for $geneM \in pxMask$ do
                for $geneO \in bitsFormingBB$ do
                    if ¬Dep($empVIG$, $geneM$, $geneO$) then
                        DLED($\boldsymbol{x_1}$, $geneM$, $geneO$);
                        if ¬Dep($empVIG$, $geneM$, $geneO$) then
                            DLED($\boldsymbol{x_2}$, $geneM$, $geneO$);
        else
            return $2, \boldsymbol{x_2'}$;
    else
        return $1, \boldsymbol{x_1'}$;
    if $f(\boldsymbol{x_1'}) = f(\boldsymbol{x_1})$ then
        return $0, \boldsymbol{x_1'}$;
    return $-1, \boldsymbol{x_1}$;
```

In the pseudocode 2, we present the proposed approach in comparison with the original implementation (pseudocode 1). Whenever we detect that we are missing a linkage from the VIG, then we collect all bits that should be tested for a direct linkage with at least one bit from *pxMask*. These are the bits where $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ differ in value and are not in *pxMask* (line 6). Using the *GetBitsFormBB* function, we get the bits that, together with the bits from *pxMask*, form BB (line 7). Finally, we test these retrieved bits to see if they have a direct linkage with the individual bits of *pxMask* using DLED, similar to the original implementation (lines 8 to 13). In the following section, we describe how these bits forming the BB are obtained.

5.2.3 Identification of Building Block with greedy Linkage Identification by Nonlinearity Check

The following procedure seeks to obtain the subset of bits that, together with the bits from *pxMask*, form the BB. Note also that this subset should be as tightly linked as possible to the bits from *pxMask*, as these bits will be tested for direct linkage. By tightness we mean a high epistasis measure caused by perturbation between subsets of bits, similar to what we described for the LIEM (3.4). This property is discussed in more detail in the section explaining the gLINC procedure (5.2.4).

At the beginning of the procedure, we are given a subset of bits to be tested for linkage with bits from *pxMask*. The order of these bits (variables) is randomly shuffled because we cannot assume that the dependent bits should be ordered in any way. (line 2) Next, a set of bits is prepared that accumulates bits tightly linked to bits from *pxMask* and a queue that will be used to search for the linked blocks. (line 3 and 4) Starting with the entire block of bits specified as *bitsToBeTested* at the beginning, we push the entire block to the queue. After this initialization, we repeat the following steps until the queue is empty. At each iteration, we check if the subset of bits is only a single-element set, if so, we join this subset with the tightly linked bits found so far. (line 7 and 8) If the subset is larger, then this subset is split into two parts of equal size. (line 10) These split parts are then separately checked using the gLINC procedure to see if they form a BB with the bits from *pxMask*. If they form a BB, then the search continues with this subset, which is then pushed to the queue. (lines 11 to 16) Otherwise, we discard this subset from the search. When this search procedure is complete, we return the collected set of bits that are tightly linked to the *pxMask* bits. (line 17)

Pseudocode 3 Detection of BB with gLINC

```
1: function GetBitsFormBB($\boldsymbol{x_1}$, $\boldsymbol{x_2}$, $bitsToBeTested$, $pxMask$)
2:     shuffleBits($bitsToBeTested$);
3:     $bitsFormingBB \leftarrow \emptyset$
4:     $queue \leftarrow <empty>$; $queue.push(bitsToBeTested)$;
5:     while $\neg queue.empty()$ do
6:         $bitsSubset \leftarrow queue.front()$; $queue.pop()$;
7:         if $bitsSubset.size() = 1$ then
8:             $bitsFormingBB \leftarrow (bitsFormingBB \cup bitsSubset)$;
9:         else
10:            $bitsFirstHalf, bitsSecHalf \leftarrow$ splitSet($bitsSubset$);
11:            if gLINC($\boldsymbol{x_1}$, $bitsFirstHalf$, $pxMask$) $\vee$
12:                gLINC($\boldsymbol{x_2}$, $bitsFirstHalf$, $pxMask$) then
13:                $queue.push(bitsFirstHalf)$;
14:            if gLINC($\boldsymbol{x_1}$, $bitsSecHalf$, $pxMask$) $\vee$
15:                gLINC($\boldsymbol{x_2}$, $bitsSecHalf$, $pxMask$) then
16:                $queue.push(bitsSecHalf)$;
17:     return $bitsFormingBB$;
```

5.2.4 Linkage Epistasis Measure

In this section, we describe how gLINC was used in the detection of BB. This procedure was originally introduced in [9] by Coffin and Clack. We have already described that procedure in Chapter 2 (3.2.2), where we showed its functionality with an example of BB detection on a single deceptive trap function. In contrast to the original case, we must also consider the case where the fitness function is composed of several overlapping subfunctions. So the fitness growth caused by the perturbation need not be strictly non-linear as in the original formulation ($\Delta f_{IJ}(x) \neq (\Delta f_I(x) + \Delta f_J(x))$). In the following pseudocode, we show how the BB identification is done.

Pseudocode 4 Greedy Linkage Identification By Nonlinearity Check (gLINC)

```
function gLINC($\boldsymbol{x}$, $bitsI$, $bitsJ$)
    $\boldsymbol{x}_I \leftarrow \boldsymbol{x}$; $\boldsymbol{x}_J \leftarrow \boldsymbol{x}$; $\boldsymbol{x}_{IJ} \leftarrow \boldsymbol{x}$;
    for $bitI \in bitsI$ do
        $\boldsymbol{x}_I[bitI] \leftarrow \neg\boldsymbol{x}[bitI]$;
        $\boldsymbol{x}_{IJ}[bitI] \leftarrow \neg\boldsymbol{x}[bitI]$;
    for $bitJ \in bitsJ$ do
        $\boldsymbol{x}_J[bitJ] \leftarrow \neg\boldsymbol{x}[bitJ]$;
        $\boldsymbol{x}_{IJ}[bitJ] \leftarrow \neg\boldsymbol{x}[bitJ]$;
    $\Delta_I \leftarrow f(\boldsymbol{x}_I) - f(\boldsymbol{x})$;
    $\Delta_J \leftarrow f(\boldsymbol{x}_J) - f(\boldsymbol{x})$;
    $\Delta_{IJ} \leftarrow f(\boldsymbol{x}_{IJ}) - f(\boldsymbol{x})$;
    if $|\Delta_{IJ} - (\Delta_I + \Delta_J)| > \delta$ then
        return true;
    else
        return false;
```

At the beginning, we prepare three copies of the given individual (solution) $\boldsymbol{x}$. (line 2) In $\boldsymbol{x_I}$ we perturb all the bits from the set *bitsI* and in $\boldsymbol{x_J}$ we perturb the bits from the set *bitsJ*. In $\boldsymbol{x_{IJ}}$ we perturb both. (lines 3 to 8) Next, we compute the fitness growth caused by these perturbations. (lines 9 to 11) If the inequality $|\Delta_{IJ} - (\Delta_I + \Delta_J)| > \delta$ holds, then we consider the sets *bitsI* and *bitsJ* as a single BB. (line 13) Otherwise, we did not find an evidence of dependence between these blocks. The inequality on line 12 replaces the original inequality $(\Delta_{IJ} \neq (\Delta_I + \Delta_J))$ from [9]. The correct setting of the δ parameter is discussed below.

The δ parameter determines the measure of nonlinearity, which is then considered as a dependency between blocks of bits. To set this parameter properly, we used a similar approach to that seen in the LIEM [11]. Moreover, in addition to the aforementioned VIG, which stores only the direct interactions between variables, we maintain one additional graph that maintains further information about the interactions, the epistasis measure. The vertices of this graph again represent the individual variables (bits) of the solution, and the edges of the graph represent the epistasis measure between the variables, which is calculated as follows:

$$\begin{aligned} \boldsymbol{x}_{\bar{i}} &= x_1 \cdots \bar{x}_i \cdots x_n \\ \boldsymbol{x}_{\bar{j}} &= x_1 \cdots \bar{x}_j \cdots x_n \\ \boldsymbol{x}_{\bar{i},\bar{j}} &= x_1 \cdots \bar{x}_i \cdots \bar{x}_j \cdots x_n \end{aligned} \tag{5.1}$$

Now with these perturbed solutions $\boldsymbol{x}_{\bar{i}}$, $\boldsymbol{x}_{\bar{j}}$ and $\boldsymbol{x}_{\bar{i},\bar{j}}$ we compute the fitness differences caused by the perturbations.

$$\begin{aligned} \Delta_i &= f(\boldsymbol{x}_{\bar{i}}) - f(\boldsymbol{x}) \\ \Delta_j &= f(\boldsymbol{x}_{\bar{j}}) - f(\boldsymbol{x}) \\ \Delta_{i,j} &= f(\boldsymbol{x}_{\bar{i},\bar{j}}) - f(\boldsymbol{x}) \end{aligned} \tag{5.2}$$

Finally, the epistasis measure between the i-th and j-th variable is given as follows:

$$e_{i,j} = |\Delta_{i,j} - (\Delta_i + \Delta_j)| \tag{5.3}$$

For each pair of variables, we update this epistasis measure during the optimization, keeping the maximum until we detect a direct linkage between these two variables. The graph storing these epistasis measures is updated from DLED, so it does not cost us any additional FFE. The δ parameter used in gLINC is updated whenever a direct linkage is detected, δ is updated as follows. We compute the average epistasis measure of those pairs of variables that are directly linked. Let us denote this average as $\bar{e}_{dep}$. The δ parameter is then updated as follows.

$$\delta = \frac{\bar{e}_{dep}}{2} \tag{5.4}$$

Thus, if the epistasis measure in gLINC is lower than this delta, we find no evidence that the blocks of variables form a single BB. Otherwise, we consider them as potentially forming a single BB. To clarify the δ parameter setting, note that if the epistasis measure between two variables ($|\Delta_{i,j} - (\Delta_i + \Delta_j)|$) is close to zero, then we consider the variables (or blocks of variables) as independent (not forming a single BB). Thus, we used the values of the epistasis measures of those pairs of variables that are guaranteed to be dependent, and estimated the level of epistasis measure that we should consider to exhibit dependence. This delta parameter is then used in gLINC to determine whether or not two blocks of variables form a single BB. If the perturbations of the blocks of variables had a linear or nearly linear effect on fitness, i.e, no nonlinearity occurred, and thus the value of $|\Delta_{IJ} - (\Delta_I + \Delta_J)|$ is "closer" to zero, then we found no evidence that the two blocks form a single BB. Otherwise, some nonlinearity has occurred, the value of $|\Delta_{IJ} - (\Delta_I + \Delta_J)|$ is "closer" to $\bar{e}_{dep}$, and therefore we consider that these blocks form a single BB. Finally, we should note that the δ parameter is initialized as 0 because we want to trigger as much linkage exploration as possible at the beginning to get at least some empirical estimate on the value of δ. This whole idea is also similar to the procedure used in [14], where the *k-means* algorithm is used to distinguish between correct and incorrect linkages.

In the following pseudocode 5, we present a modified DLED procedure with the update of the epistasis measure (EM).

The procedure is similar to that in pseudocode 4, but here we only perturb individual bits, not whole blocks. As with gLINC, we prepare three copies of individual $\boldsymbol{x}$. In each of them, we flip the i-th, j-th, and both bits, respectively. (lines 3 to 8) We then calculate the fitness differences caused by the perturbations (lines 10 to 12). We update the epistasis measure for the i-th and j-th bits similarly to the LIEM[11] procedure described in 3.4. (lines 13 and 14) Finally, if the condition on line 16 holds, we find that the i-th bit is directly dependent on the j-th bit, so we update the VIG accordingly (line 17) and recompute the δ (line 18).

Pseudocode 5 Direct Linkage Empirical Discovery (DLED) with EM Update

```
function DLED($\boldsymbol{x}$, $bit_i$, $bit_j$)
    $\boldsymbol{x}_i \leftarrow \boldsymbol{x}$; $\boldsymbol{x}_j \leftarrow \boldsymbol{x}$; $\boldsymbol{x}_{ij} \leftarrow \boldsymbol{x}$;
    // flip ith bit
    $\boldsymbol{x}_i[bit_i] \leftarrow \neg\boldsymbol{x}[bit_i]$;
    $\boldsymbol{x}_{ij}[bit_i] \leftarrow \neg\boldsymbol{x}[bit_i]$;
    // flip jth bit
    $\boldsymbol{x}_j[bit_j] \leftarrow \neg\boldsymbol{x}[bit_j]$;
    $\boldsymbol{x}_{ij}[bit_j] \leftarrow \neg\boldsymbol{x}[bit_j]$;
    // update EM
    $\Delta_i \leftarrow f(\boldsymbol{x}_i) - f(\boldsymbol{x})$;
    $\Delta_j \leftarrow f(\boldsymbol{x}_j) - f(\boldsymbol{x})$;
    $\Delta_{ij} \leftarrow f(\boldsymbol{x}_{ij}) - f(\boldsymbol{x})$;
    $emMat[bit_i][bit_j] \leftarrow \max(emMat[bit_i][bit_j], |\Delta_{ij} - (\Delta_i + \Delta_j)|)$;
    $emMat[bit_j][bit_i] \leftarrow \max(emMat[bit_j][bit_i], |\Delta_{ij} - (\Delta_i + \Delta_j)|)$;
    // update VIG
    if $(f(\boldsymbol{x}) < f(\boldsymbol{x}_j)) \neq (f(\boldsymbol{x}_i) < f(\boldsymbol{x}_{ij}))$ then
        addDepToVIG($bit_i$, $bit_j$);
        recomputeDelta();
```

5.3 Greedy Linkage Learning Features

In this section we would like to summarize the features of the presented approach.

In the above procedure of discarding some linkages from further testing, we still kept the original property of dgGA, that we trigger interaction learning only when a linkage is missing. This discarding of some linkages (pair of bits, potentially dependent) should have a positive effect on linkage learning efficiency. That is, we consume less FFE purely on the linkage learning. These savings arise from the fact that at the beginning of learning new linkages, before we test the direct linkage between bits, we directly discard the entire block of bits that are independent of the bits for which we are looking for new linkages. We believe that these savings should be especially beneficial on large problems, those with a large number of variables (bits), and also on problems where each variable is directly dependent on only a small number of others.

In order to properly estimate the parameter that determines when two blocks of variables are no longer dependent, we need to maintain another graph (matrix) in addition to VIG, in which we store the epistasis measure between the individual variables. This graph is updated whenever we run the DLED procedure and thus does not cost any additional FFE.

Overall, the modified optimizer remains, like the original dgGA, parameterless.

Chapter 6

Experiments

6.1 Experiment setup

We considered a set of five different problems of various lengths, namely MAX-3SAT and Ising Spin Glass, as representatives of natural pseudo-boolean optimization problems, as well as a concatenation of bimodal and deceptive trap functions (with different overlaps) as a representative of the Mk Landscapes problem, which are often used as benchmark problems for GAs. For testing purposes, we consider the same test instances as in [2].

For MAX-3SAT problems, we search for a variable assignment that satisfies as many clauses as possible, similarly as in [2], we only consider cases where all clauses are satisfiable. The MAX-3SAT problem was already described in the first chapter (2.1.2). For MAX-3SAT, the strength of the overlap, the occurrence of the same variable in multiple clauses, is given by the *clause ratio* (CR) parameter. The CR parameter is defined as the ratio between the number of clauses and the number of variables.

We have described the ISG problem in the example of the DLED (4.3.1) procedure. We considered three different lengths of this problem that were solvable by the original dgGA, since we do not assume that our modification of dgGA could solve those cases that remained unsolvable by the original dgGA.

The problem denoted below as Mk-deceptive is a concatenation of the deceptive trap function, which has already been described in the example of the missing linkage procedure (4.2.1). Thus, here we consider a concatenation of M deceptive trap functions, each dependent on k variables, and these subfunctions overlap by o variables. The M denotes the number of blocks. The problem denoted as Mk-bimodal is a concatenation of order-k bimodal trap functions, which is defined as follows:

$$f_{bimodalTrap}(\boldsymbol{x}) = \begin{cases} \frac{k}{2} - |u(\boldsymbol{x}) - \frac{k}{2}| - 1 & , u(\boldsymbol{x}) \neq k \wedge u(\boldsymbol{x}) \neq 0 \\ \frac{k}{2} & , u(\boldsymbol{x}) = k \vee u(\boldsymbol{x}) = 0 \end{cases} \tag{6.1}$$

Where u is the unitation function and is defined as the number of ones in $\boldsymbol{x}$, recall also that k denotes the length of the function. The main difference between the bimodal and the deceptive function is that the bimodal has two global optima and

$\binom{k}{k/2}$ local optima, and therefore it is more difficult to detect linkages, which was discussed in [22].

In order to obtain accurate results and, more importantly, to verify the effectiveness of the proposed changes to dgGA, we used the implementation from the authors of dgGA. This implementation is publicly available on GitHub repository[1]. We reimplemented modified parts of dgGA to use gLINC in the linkage learning procedure, all coded in C++. The experiments were performed on two different computers, one with the Intel Core i7-4702MQ CPU 2.20GHz 16GB RAM and second with the Intel Core i5-6500 CPU 3.20GHz 16GB RAM, both with Windows 10 installed. Since we were primarily interested in the number of FFEs, this choice should not affect the correctness of the results. In the overall comparison, we also show the time required to obtain the optimal solution. These times are only approximate. Therefore, these values should be taken with a grain of salt. We labeled these times in the table with ♣ if this experiment was done on the first computer, otherwise with ♢ if it was done on the second computer. Furthermore, we performed an experiment to obtain at least an approximate constant for converting the times between these computers. Thus the recalculation of these times is given approximately as follows:

$$t_{PC_1} = 1.72 t_{PC_2} \tag{6.2}$$

For each of the above mentioned problems, we consider a set of ten different problem instances. We repeated each experiment fifteen times, also using a time limit for the computation. The experiment was terminated after finding the optimal solution or after 8 hours of computation.

The repository mentioned above also contains the problem definition files that were used for the experiments. We did not change any settings of dgGA (initial population optimizer, crossover for sliding individuals, etc.) when we performed the experiments.

In addition to the number of FFEs, we also use the *Fill* measure to compare the quality of the linkage learning, which was also used in [2] and introduced in [22]. The *Fill* value is defined as the median of the percentage of true direct linkages found for each bit. Thus, $0 \leq \mathit{Fill} \leq 1$, and if *Fill* is 1, it means that all linkages have been found for each bit.

In the results below, the original version of dgGA is labeled as dgGA-DLED, and the version with our modification is labeled as dgGA-gLINC-DLED.

6.2 Results and discussion

In the following table 6.1, we present the results of the quality of linkage learning and cost. The number of FFEs here refers only to the FFEs that were spent on the linkage learning itself; in addition, we report the percentage of total FFEs. The value of *Fill* should be understood as we stated above.

[1]https://github.com/przewooz/darkGrayGA

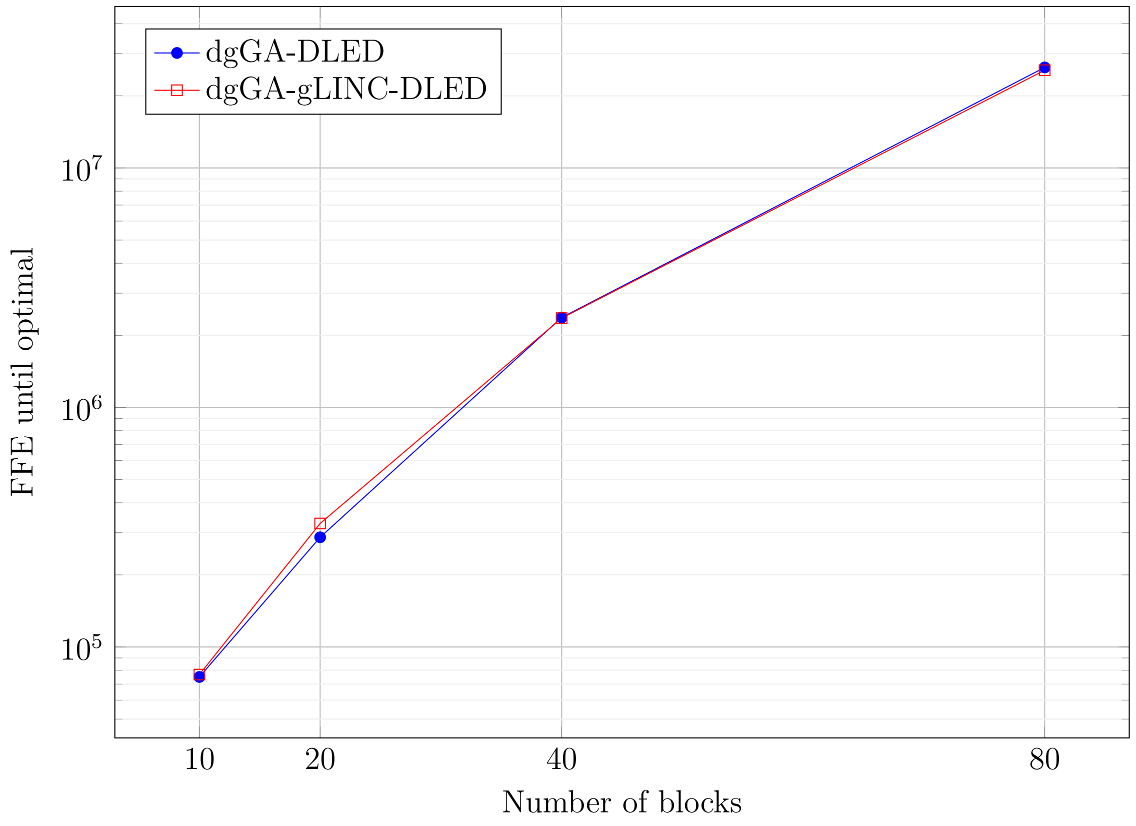
From the results, it can be concluded that the changes we made to the dgGA do not affect the quality of linkage learning. There is only a deterioration in the linkage learning quality for MAX-3SAT and Mk-deceptive(k=5, o=3, size=40 blocks), but this deterioration is so small that we believe it is almost indistinguishable.

Regarding the spending of FFEs on the linkage learning itself, it seems that our modification of dgGA seems to have achieved success mainly on those problems with many variables. For these large ones, in cases where the original dgGA algorithm spent a large percentage of FFEs on learning interactions, we were able to reduce this percentage. For the ISG problem, due to the modification made in dgGA, this percentage was reduced rapidly. For the Mk-bimodal problems by roughly half. For the Mk-deceptive problems, in case of Mk-deceptive(k=5, o=3), the percentage was able to be reduced only for large instances, while in case of Mk-deceptive(k=8, o=5), in fact, there was not much to improve since the expenditure on linkage learning was already low in the original dgGA.

Table 6.1: Empirical linkage quality and generation cost comparison. Number of fitness function evaluations (FFEs) spent only on linkage detection in the median run (median number of total FFEs) together with the percentage of total FFEs needed to obtain the optimal solution. And the corresponding quality of the fitness function decomposition is shown via the Fill measure.

Problem	Size	dgGA-DLED			dgGA-gLINC-DLED		
		[FFE]	[%]	[Fill]	[FFE]	[%]	[Fill]
MAX-3SAT (CR=4.27)	50 bits	2.25E+04	21.17	0.85	2.16E+04	19.79	0.84
ISG	484 bits	1.25E+06	24.87	1.00	1.12E+05	2.93	1.00
	625 bits	2.32E+06	26.05	1.00	1.47E+05	2.16	1.00
	784 bits	3.78E+06	26.27	1.00	1.98E+05	1.83	1.00
Mk-deceptive (k=8, o=5)	10 blocks	1.38E+03	1.84	0.97	3.05E+03	3.97	0.97
	20 blocks	7.25E+03	3.79	0.96	4.39E+04	13.38	0.97
	40 blocks	2.7E+04	1.14	0.94	1.89E+04	0.8	0.97
	80 blocks	5.86E+04	0.22	0.96	4.46E+04	0.17	0.97
Mk-deceptive (k=5, o=3)	10 blocks	2.1E+03	39.24	0.94	4.58E+03	57.51	0.95
	20 blocks	9.09E+03	47.31	0.93	1.23E+04	54.65	0.96
	40 blocks	1.59E+04	20.49	0.96	2.92E+04	39.6	0.95
	80 blocks	1.31E+05	46.11	0.95	7.54E+04	31.82	0.95
	120 blocks	2.27E+05	25.28	0.96	6.4E+04	9.59	0.96
Mk-bimodal (k=10, o=5)	10 blocks	2.01E+06	34.78	1.00	5.58E+05	15.16	1.00
	20 blocks	2.15E+07	32.22	1.00	4.25E+06	17	1.00

Figure 6.1: FFE-based scalability on the Mk-deceptive (k=8, o=5) problem



In Figure 6.1 we present the FFE-based scalability on the Mk-deceptive (k=8, o=5) problem. Overall, we can say that in this case the performance of the optimizer has not changed much due to our modification, reaching the optimum after almost the same number of FFEs. We assume that this is due to the fact that already in the original dgGA the cost of learning the interactions was low, so in this case our modification does not have much to save. However, if we examine the results in more detail, we can notice that our modification of dgGA is a tiny bit worse for small instances and, on the contrary, a tiny bit better for large ones. This is due to the fact that, in general, our modification should work better for large instances, since here we are more likely to save the number of FFEs for learning interactions and thus the total number of FFEs to reach the optimum.

In Figure 6.2 we present the FFE-based scalability on the Mk-deceptive (k=5, o=3) problem. As can be inferred from this figure, our modification in dgGA performs better for larger instances of the problem being solved, and worse for smaller ones. We assume that for those instances where it performed worse, this is because here our modification in dgGA spent more FFEs on learning the interactions and thus the total number of FFEs is higher. Conversely, for the larger instances, we were able to reduce the number of potential linkages and thus save the expense on linkage learning, which then resulted in a lower total number of FFEs.

In the last Figure 6.3, we present the FFE-based scalability on the Ising Spin Glass problem. For all considered lengths of this problem, our modification of dgGA resulted in an improvement in the total number of FFEs. We believe this is due to the fact that, in general, for this problem the number of variables is high while

Figure 6.2: FFE-based scalability on the Mk-deceptive (k=5, o=3) problem

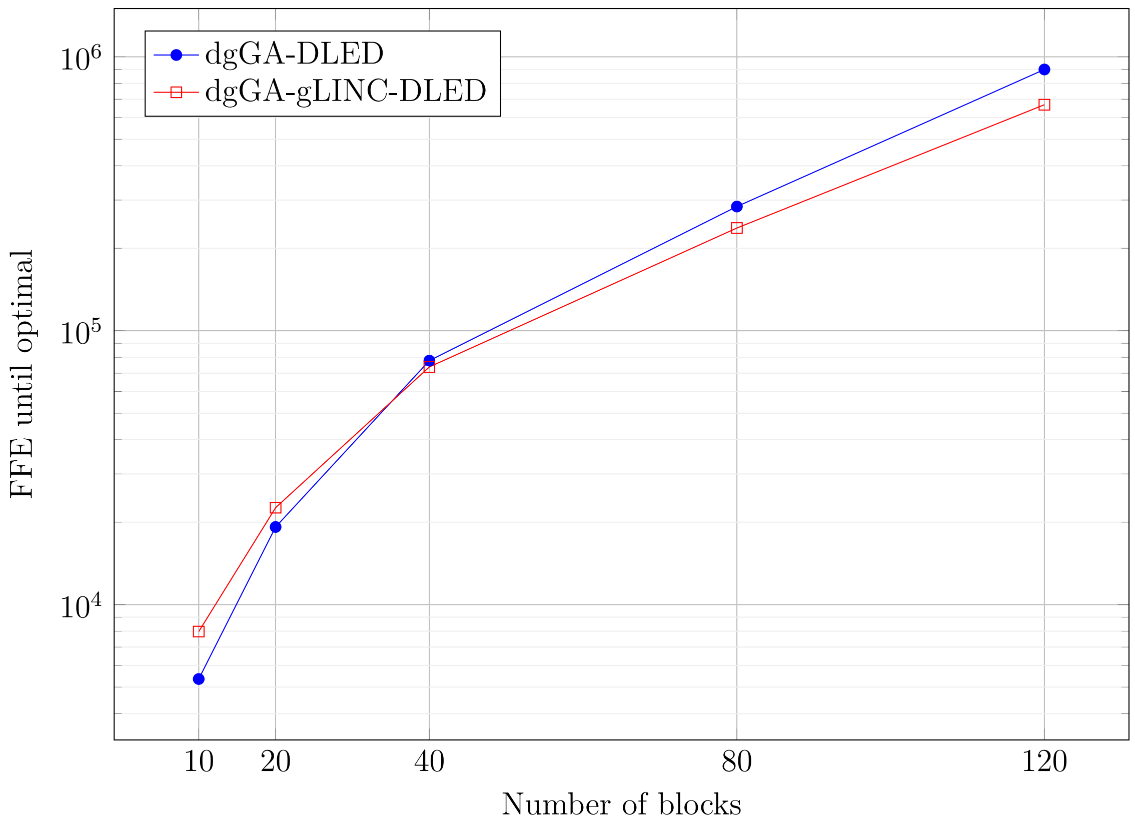
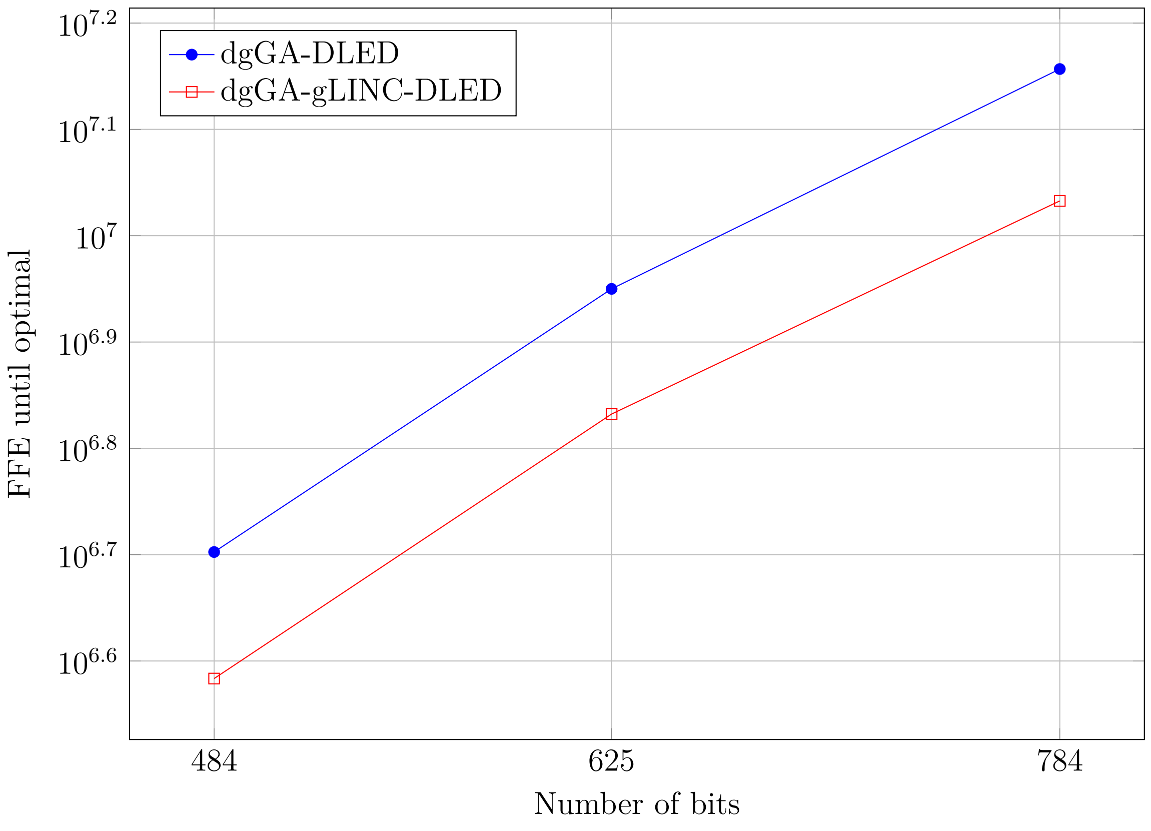


Figure 6.3: FFE-based scalability on the Ising Spin Glass problem



the number of direct dependencies of each variable remains low. (Recall that in the ISG problem, each variable is directly dependent on at most four others.) Thus, the reduction of the tested interactions, here, successfully reduced the initially large number of potentially dependent variables to only a small fraction of these variables, thus saving the testing of direct linkages that would have been tested unnecessarily. This subsequently had a further impact on the total number of FFEs.

Table 6.2 presents the overall results for all problems considered. In this table, in addition to the median value, we also report the Interquartile Range (IQR) and the time required to obtain the optimal solution. The ISG and Mk-deceptive problems have already been discussed above. For the MAX-3SAT problem, we obtained a similar result after modification in dgGA; but again, there was not much FFE to be saved. Longer MAX-3SAT instances than those listed in the table failed to be solved as in the original dgGA. For the Mk-bimodal problem, in both cases of the length considered, the modifications done in the dgGA resulted in reducing the total number of FFEs needed to find the optimal solution. This reduction can be reasoned by the saving of the FFEs needed to learn the interactions.

Table 6.2: General comparison. Total number of fitness function evaluations (FFEs) needed to obtain the optimal solution together with the Interquartile Range (IQR) and approximate time in seconds to obtain the optimal solution. Values are marked with ♣ if the experiment was performed on the first computer and with ♢ if it was performed on the second computer.

		dgGA-DLED			dgGA-gLINC-DLED		
Problem	Size	Total FFE [FFE]	IQR	Time [s]	Total FFE [FFE]	IQR	Time [s]
MAX-3SAT (CR=4.27)	50 bits	1.06E+05	1.74E+05	1.34♣	1.09E+05	3.65E+05	0.8♢
ISG	484 bits	5.05E+06	3.32E+06	153♢	3.84E+06	3.32E+06	345♣
	625 bits	8.92E+06	4.82E+06	375♢	6.8E+06	2.15E+06	633♣
	784 bits	1.44E+07	1.71E+06	746♢	1.08E+07	1.51E+06	701♢
Mk-deceptive (k=8, o=5)	10 blocks	7.5E+04	3.74E+04	0.24♢	7.67E+04	3.65E+04	0.28♣
	20 blocks	2.87E+05	2.85E+05	1.87♢	3.28E+05	2.86E+05	3.46♣
	40 blocks	2.37E+06	2.34E+06	35.8♢	2.36E+06	2.36E+06	52.7♣
	80 blocks	2.63E+07	2.59E+07	1812♢	2.56E+07	2.76E+07	1298♢
Mk-deceptive (k=5, o=3)	10 blocks	5.35E+03	3.1E+03	0.01♢	7.98E+03	4.3E+03	0.02♣
	20 blocks	1.92E+04	9.6E+03	0.05♢	2.26E+04	1.37E+04	0.08♣
	40 blocks	7.78E+04	3.3E+04	0.55♢	7.73E+04	2.99E+04	0.63♣
	80 blocks	2.84E+05	2.33E+05	3.09♢	2.37E+05	1.58E+05	5.99♣
	120 blocks	8.99E+05	1.78E+06	27.1♢	6.68E+05	5.09E+05	20.2♢
Mk-bimodal (k=10, o=5)	10 blocks	5.78E+06	5.95E+06	49.9♢	3.68E+06	3.88E+06	27.7♢
	20 blocks	6.68E+07	6.5E+07	688♢	2.5E+07	2.2E+07	345♢

Chapter 7

Conclusion

In this thesis, we have introduced the modification of the procedure that learns the interactions among the variables in dgGA. We proposed this modification based on a previous analysis of this algorithm and also by studying known techniques for identifying linkages among variables using perturbation. To employ gray box optimization techniques, knowledge of the direct dependencies among variables is required. In dgGA, the original learning method satisfies all the required properties, detects only direct linkages and never detects false linkages. Therefore, we did not replace the original linkage learning method completely, but we came up with an idea to handle the overall linkage learning among variables more efficiently. We replaced the original learning method, that tested dependencies one by one, with a modified procedure that discards entire independent blocks. We implemented the modification in dgGA and then verified it on the MAX-3SAT problem, Ising Spin Glass, and concatenation of deceptive and bimodal trap functions compared to the original version. Based on the results, we conclude that the modification

- does not degrade the quality of the problem decomposition,
- reduces the number of function evaluations spent on interaction learning for large instances of tested problems, and thus
- improves the efficiency of the optimizer for larger instances.

However, we must also admit that for small instances, our modification resulted in higher learning costs.

Bibliography

1. WHITLEY, L. Darrell; CHICANO, Francisco; GOLDMAN, Brian W. Gray Box Optimization for Mk Landscapes Nk Landscapes and Max-Ksat. *Evol. Comput.* 2016, vol. 24, no. 3, pp. 491–519. ISSN 1063-6560. Available from DOI: `10.1162/EVCO_a_00184`.
2. PRZEWOZNICZEK, Michal W.; TINÓS, Renato; FREJ, Bartosz; KOMARNICKI, Marcin M. On Turning Black - into Dark Gray-Optimization with the Direct Empirical Linkage Discovery and Partition Crossover. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Boston, Massachusetts: Association for Computing Machinery, 2022, pp. 269–277. GECCO '22. ISBN 9781450392372. Available from DOI: `10.1145/3512290.3528734`.
3. DUSHATSKIY, Arkadiy; VIRGOLIN, Marco; BOUTER, Anton; THIERENS, Dirk; BOSMAN, Peter A. N. Parameterless Gene-pool Optimal Mixing Evolutionary Algorithms. *CoRR*. 2021, vol. abs/2109.05259. Available from arXiv: `2109.05259`.
4. BOROS, Endre; HAMMER, Peter L. Pseudo-Boolean optimization. *Discrete Applied Mathematics*. 2002, vol. 123, no. 1, pp. 155–225. ISSN 0166-218X. Available from DOI: `https://doi.org/10.1016/S0166-218X(01)00341-9`.
5. ALARIE, Stéphane; AUDET, Charles; GHERIBI, Aïmen E.; KOKKOLARAS, Michael; LE DIGABEL, Sébastien. Two decades of blackbox optimization applications. *EURO Journal on Computational Optimization*. 2021, vol. 9, pp. 100011. ISSN 2192-4406. Available from DOI: `https://doi.org/10.1016/j.ejco.2021.100011`.
6. SOLNIK, Benjamin; GOLOVIN, Daniel; KOCHANSKI, Greg; KARRO, John Elliot; MOITRA, Subhodeep; SCULLEY, D. Bayesian Optimization for a Better Dessert. In: *Proceedings of the 2017 NIPS Workshop on Bayesian Optimization*. December 9, 2017, Long Beach, USA, 2017. The workshop is BayesOpt 2017 NIPS Workshop on Bayesian Optimization December 9, 2017, Long Beach, USA.
7. TINÓS, Renato; WHITLEY, Darrell; CHICANO, Francisco. Partition Crossover for Pseudo-Boolean Optimization. In: *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*. Aberystwyth, United Kingdom: Association for Computing Machinery, 2015, pp. 137–149. FOGA '15. ISBN 9781450334341. Available from DOI: `10.1145/2725494.2725497`.

8. MUNETOMO, M.; GOLDBERG, D.E. A genetic algorithm using linkage identification by nonlinearity check. In: *IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.99CH37028)*. 1999, vol. 1, 595–600 vol.1. ISSN 1062-922X. Available from DOI: 10.1109/ICSMC.1999.814159.

9. COFFIN, David J.; CLACK, Christopher D. GLINC: Identifying Composability Using Group Perturbation. In: *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*. Seattle, Washington, USA: Association for Computing Machinery, 2006, pp. 1133–1140. GECCO '06. ISBN 1595931864. Available from DOI: 10.1145/1143997.1144178.

10. MUNETOMO, Masaharu; GOLDBERG, David E. Linkage Identification by Non-Monotonicity Detection for Overlapping Functions. *Evol. Comput.* 1999, vol. 7, no. 4, pp. 377–398. ISSN 1063-6560. Available from DOI: 10.1162/evco.1999.7.4.377.

11. MUNETOMO, M. Linkage identification based on epistasis measures to realize efficient genetic algorithms. In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*. 2002, vol. 2, 1332–1337 vol.2. Available from DOI: 10.1109/CEC.2002.1004436.

12. DAVIDOR, Yuval. Epistasis Variance: Suitability of a Representation to Genetic Algorithms. *Complex Syst.* 1990, vol. 4.

13. TSUJI, Miwako; MUNETOMO, Masaharu; AKAMA, Kiyoshi. Linkage Identification by Fitness Difference Clustering. *Evol. Comput.* 2006, vol. 14, no. 4, pp. 383–409. ISSN 1063-6560. Available from DOI: 10.1162/evco.2006.14.4.383.

14. HELMI, B. Hoda; PELIKAN, Martin; RAHMANI, Adel T. Linkage Learning Using the Maximum Spanning Tree of the Dependency Graph. In: *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 2012, pp. 1429–1430. GECCO '12. ISBN 9781450311786. Available from DOI: 10.1145/2330784.2330970.

15. CHUANG, Chung-Yao; CHEN, Ying-ping. Likage identification by perturbation and decision tree induction. In: *2007 IEEE Congress on Evolutionary Computation*. 2007, pp. 357–363. ISSN 1941-0026. Available from DOI: 10.1109/CEC.2007.4424493.

16. QUINLAN, J. R. Induction of decision trees. *Machine Learning*. 1986, vol. 1, no. 1, pp. 81–106. ISSN 1573-0565. Available from DOI: 10.1007/BF00116251.

17. PRZEWOZNICZEK, Michal W.; KOMARNICKI, Marcin M.; FREJ, Bartosz. Direct Linkage Discovery with Empirical Linkage Learning. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Lille, France: Association for Computing Machinery, 2021, pp. 609–617. GECCO '21. ISBN 9781450383509. Available from DOI: 10.1145/3449639.3459333.

18. PRZEWOZNICZEK, Michal W.; KOMARNICKI, Marcin M. Empirical Linkage Learning. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. Cancún, Mexico: Association for Computing Machinery, 2020, pp. 23–24. GECCO '20. ISBN 9781450371278. Available from DOI: 10.1145/3377929.3397771.

19. WIKIPEDIA. *Hamming distance — Wikipedia, The Free Encyclopedia* [http://en.wikipedia.org/w/index.php?title=Hamming%20distance&oldid=1148896017]. 2023. [Online; accessed 11-April-2023].

20. GOLDMAN, Brian W.; PUNCH, William F. Parameter-Less Population Pyramid. In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 785–792. GECCO '14. ISBN 9781450326629. Available from DOI: 10.1145/2576768.2598350.

21. WIKIPEDIA. *Divide-and-conquer algorithm — Wikipedia, The Free Encyclopedia* [http://en.wikipedia.org/w/index.php?title=Divide-and-conquer%20algorithm&oldid=1137028109]. 2023. [Online; accessed 08-May-2023].

22. PRZEWOZNICZEK, Michal W.; FREJ, Bartosz; KOMARNICKI, Marcin M. On Measuring and Improving the Quality of Linkage Learning in Modern Evolutionary Algorithms Applied to Solve Partially Additively Separable Problems. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. Cancún, Mexico: Association for Computing Machinery, 2020, pp. 742–750. GECCO '20. ISBN 9781450371285. Available from DOI: 10.1145/3377930.3390242.

Appendix

A Implementation of the Dark Gray Genetic Algorithm with Greedy Linkage Learning

List of attachments

- `darkGrayGa.cpp`
- `darkGrayGa.h`

Sources code of the implementation of the modified dgGA with Greedy Linkage Learning.

To run the modified dgGA with Greedy Linkage Learning, download the source files of the original dgGA from the publicly available Github repository[1] and replace the original `darkGrayGa.cpp` and `darkGrayGa.h` files with our modified ones before build. To set up the project correctly, follow the instructions in the readme of the original implementation. Note here that Visual Studio 2017 is required to build the project.

The main modification of dgGA was made in the following list of methods in the `darkGrayGa.cpp` file, which also includes the line where the implementation of the corresponding method starts.

- `void nDarkGrayGA::CDarkGA_DSM::vRecompLIEMCentrs();` (line 1884)
- `void nDarkGrayGA::CDarkGA_DSM::vShuffleVars(vector<int> & vec_vars);` (line 1900)
- `bool nDarkGrayGA::CDarkGA_DSM::b_glinc(vector<int> & vecGenesI,`
  `vector<int> & vecGenesJ,`
  `CDarkGrayGAIndividual *pcExtractionIndividual);` (line 2214)
- `void CDarkGA_DSM::vUpdateDSM(CDarkGrayGAPxMask *pcMask);` (line 1761)
- `void CDarkGA_DSM::b_extract_dled_for_gene_pair(`
  `int iGeneBlock,`
  `int iGeneContext,`

[1]https://github.com/przewooz/darkGrayGA

```
        int *piDledExtractionMask,
        CDarkGrayGAIndividual *pcExtractionIndividual,
        int *piNewPairsDetected); (line 1926)
```

B Tables with complete results

List of attachments

- `max3sat_cr4_27_vars_50.xlsx`
- `isg_484.xlsx`
- `isg_625.xlsx`
- `isg_784.xlsx`
- `mk_deceptive5_m10_o3.xlsx`
- `mk_deceptive5_m20_o3.xlsx`
- `mk_deceptive5_m40_o3.xlsx`
- `mk_deceptive5_m80_o3.xlsx`
- `mk_deceptive5_m120_o3.xlsx`
- `mk_deceptive8_m10_o5.xlsx`
- `mk_deceptive8_m20_o5.xlsx`
- `mk_deceptive8_m40_o5.xlsx`
- `mk_deceptive8_m80_o5.xlsx`
- `mk_bimodal10_m10_o5.xlsx`
- `mk_bimodal10_m20_o5.xlsx`

Tables with results of complete experiments. Each row of the table corresponds to one run of the experiment, and each fifteen consecutive rows correspond to one instance of the considered problem.