Universiteit Utrecht

Department of Information and Computing Sciences

Analysis of Schema Grammar

Master Thesis
ICA-3253473

Author:
Benjamin Burger
Utrecht University

Supervisors:
dr. ir. D. Thierens
prof. dr. ir. L.C. van der Gaag
Utrecht University

November 2015

Abstract

Recently, several variants of the Schema Grammar (SG) algorithm have been proposed. This novel algorithm learns linkage between the values of variables of a problem. The linkage information learned is based on co-occurrence of allele tuples. Several variants of SG will be tested on laboratory benchmark problems and it will be shown that the framework is effectively and efficiently able to solve the problems tested. The computational complexity of the model-building algorithm will be improved from $O(n^3 \cdot p^3)$ to $O(n^2 \cdot p \cdot \log(n \cdot p) + n \cdot p^2)$ and a new version of SG will be proposed. This version will be able to outperform the original results on all problems tested in terms of run-time and fitness function evaluations. Additionally, this new version is able to solve problems with comparable results to Linkage Tree Genetic Algorithm (LTGA) on Trap, 2D Spin Glasses and NK-landscapes. This version achieves comparable run-time scaling behaviour to LTGA. In addition the relation to other algorithms (including LTGA, Apriori and Krimp) will be explored. The latter two algorithms originate from the field of Frequent Pattern Set Mining.

Contents

1 Introduction **1**
1.1 Overview of Thesis 2

2 Background **3**
2.1 Benchmark Problems 3
2.2 Gene-pool Optimal Mixing Evolutionary Algorithms 6

3 Multi-Scale Search: Schema Grammar **13**
3.1 Schema Grammar 14
3.1.1 Formal Definition 17
3.1.2 Grammar Inference 17
3.1.3 Complexity Grammar Inference 21
3.1.4 Schema Search 28
3.1.5 Variants of Schema Grammar 29
3.1.6 Scaling of SG-Trap 34
3.2 Relation GOMEA 36
3.3 Relation Frequent Pattern-Set Mining 38
3.3.1 Apriori 38
3.3.2 Krimp 39

4 Performance Analysis **41**
4.1 Methodology 41
4.2 SG-Trap 42
4.2.1 SG-Trap: Trap Functions 43
4.2.2 Counting Ones 47
4.2.3 SG-Trap: Overlapping Sub-functions 48
4.3 SG-NK 51
4.4 Schema Grammar 53
4.5 Schema Grammar v1.1 62
4.6 SG v1.1 with FI 69

5 Future Research **76**
5.1 New Problems 76
5.2 Parameter-less Schema Grammar 77

6 Conclusion **78**

List of Figures

2.1 Example of a Trap function of n bits 4
2.2 MLNGA on Onemax, Trap, NK, and MAXCUT 12

3.1 Example of a schema hierarchy for a individual on a NK-landscape 22
3.2 Example of a balanced tree 27
3.3 Example of a fully balanced tree 27
3.4 Average Fitness function evaluations on Trap functions with SG-Trap 34
3.5 Average Fitness function evaluations on hTrap functions with SG-Trap 35
3.6 Minimal population size to model Trap with GI 35

4.1 Minimal population size to correctly model Trap function structure for GI 43
4.2 # Fitness function evaluation mean versus problem size on Trap functions using SG-Trap 45
4.3 # Fitness function evaluation mean versus problem size on hTrap functions using SG-Trap 45
4.4 Variance in # fitness function evaluations on Trap functions with SG-Trap 46
4.5 Variance in # fitness function evaluations on Trap functions with SG-Trap 46
4.6 Population size versus problem size on 2D Spin Glasses using SG-Trap 49
4.7 # Fitness function evaluation mean versus problem size on 2D Spin Glasses using SG-Trap 49
4.8 Population size versus problem size on NK-landscapes using SG-Trap 50
4.9 # Fitness function evaluation mean versus problem size on NK-landscapes using SG-Trap 50
4.10 Population size versus problem size on Trap functions using SG-NK 52

4.11 # Fitness function evaluation mean versus problem size on Trap functions using SG-NK 52
4.12 Population size versus problem size on 2D Spin Glasses using SG-NK 54
4.13 Population size versus problem size on NK-landscapes using SG-NK 54
4.14 # Fitness function evaluation mean versus problem size on 2D Spin Glasses using SG-NK 55
4.15 # Fitness function evaluation mean versus problem size on NK-landscapes using SG-NK 55
4.16 Population size versus problem size on Trap functions using SG . 57
4.17 # Fitness function evaluation mean versus problem size on Trap functions using SG 57
4.18 Population size versus problem size on 2D Spin Glasses using SG 59
4.19 # Fitness function evaluation mean versus problem size on 2D Spin Glasses using SG 59
4.20 Population size versus problem size on NK-landscapes using SG . 60
4.21 # Fitness function evaluation mean versus problem size on NK-landscapes using SG 60
4.22 Population size versus problem size on Trap functions using SG v1.1 63
4.23 # Fitness function evaluation mean versus problem size on Trap functions using SG v1.1 63
4.24 Population size versus problem size on 2D Spin Glasses using SG v1.1 65
4.25 Population size versus problem size on NK-landscapes using SG v1.1 65
4.26 # Fitness function evaluation mean versus problem size on 2D Spin Glasses using SG v1.1 66
4.27 # Fitness function evaluation mean versus problem size on NK-landscapes using SG v1.1 66
4.28 Run-time versus problem size on 2D Spin Glasses using SG v1.1 68
4.29 # Run-time versus problem size on NK-landscapes using SG v1.1 68
4.30 Population size versus problem size on Trap functions using SG v1.1 with hill-climbing to create the initial population 71
4.31 # Fitness function evaluation mean versus problem size on Trap functions using SG v1.1 with hill-climbing the initial population . 71
4.32 Population size versus problem size on 2D Spin Glasses using SG v1.1 with hill-climbing to create the initial population 73
4.33 Population size versus problem size on NK-landscapes using SG v1.1 with hill-climbing to create to create the initial population . 73
4.34 # Fitness function evaluation mean versus problem size on 2D Spin Glasses using SG v1.1 with hill-climbing to create the initial population 74

4.35 # Fitness function evaluation mean versus problem size on NK-landscapes using SG v1.1 with hill-climbing to create the initial population 74

List of Tables

4.1 Results for SG on Trap functions 56
4.2 Results for SG on 2D Spin Glasses 58
4.3 Results for SG on NK-landscapes 61
4.4 Results for SG v1.1 on Trap functions 62
4.5 Results for SG v1.1 on 2D Spin Glasses and NK-landscapes . . . 67
4.6 Results for SG v1.1 with hill-climbing on Trap functions 72
4.7 Results for SG v1.1 with hill-climbing on 2D Spin Glasses and NK-landscapes 75

Tables used for the GI example in Chapter 3 have been excluded from this list.

List of Algorithms

1 Gene-pool Optimal Mixing Evolutionary Algorithm 7
2 Grammar Inference 18
3 Schema Search 29
4 Greedy Ascent 30
5 Specialized Hill-Climb Algorithm for Trap for SG 31
6 Schema Grammar 32
7 Schema Grammar v1.1 33
8 First Improvement Hill-Climbing Algorithm 70

Chapter 1

Introduction

In the field of Black Box Optimization (BBO) researchers are looking for ways to solve problems while treating the problem as a black box. The only thing that is known of the problem is a fitness or objective function that describes how good or how bad a solution is in terms of a real number. This field has many applications in the real world on problems that are too complex to solve directly. For example, Rahat et al. have proposed an evolutionary algorithm to approximate the optimal trade-off between minimum lifetime and the average lifetime of nodes in a battery powered wireless sensor network [1].

For BBO in the discrete space, Linkage Tree Genetic Algorithm (LTGA) is currently the state-of-the-art [2] [3]. LTGA is able to solve a large range of problems from laboratory benchmark problems to NP-hard problems such as MAX-CUT [4]. LTGA is a population-based evolutionary algorithm that performs an evolutionary search guided by a Linkage Tree (LT). This linkage model encapsulates interactions between variables and uses this information to perform an evolutionary search.

Recently, a novel BBO algorithm has been proposed for the discrete space. This algorithm bears the name Schema Grammar (SG) and is an instance of Multi-Scale Search (MSS) Algorithms [5] [6] [7]. Similarly to LTGA, SG learns the linkage of a problem. SG uses an alternative approach to learn the linkage of the problem. SG learns linkage information on a value level (i.e. it learns interactions between the values of the variables instead of the interactions between the variables themselves). SG is particularly interesting to our research because this approach allows for a more complex and richer linkage models than LTGA or the Family Of Subsets (FOS) in GOMEA in general. Additionally recent results claim that this framework is efficiently able to solve Trap Functions [5] and NK-landscapes [6]. For these reasons SG can be considered an alternative approach to LTGA.

There are little results published for the variants of SG, therefore the aim

of this research is to investigate the computational-scaling properties of this framework and see if this can be improved. There have been several variants of SG published; SG-Trap is aimed at solving various Trap functions [5], SG-NK solves NK landscapes [6] and SG is an all-round BBO algorithm published in the PhD Thesis of Chris Cox [7]. To research the computational properties of this framework all variants of SG will be tested on laboratory benchmark problems. The model-building technique used in SG is named Grammar Inference (GI). Currently the data-structure used in model-building technique is not optimized for frequent used operations. We will investigate different implementations for GI and alternatives to GI investigated. In the field of Frequent Pattern-Set Mining (FPSM) interesting patterns are extracted from databases. This is precisely what GI does, for this reason the relation between GI and algorithms originating from FPSM (including Krimp [8]) will be explored.

We will propose a k-ary trees data-structure to be used in GI. This data-structure improves the complexity of the model-building technique from $O^*(n^3 \cdot p^2)$ to $O(n^2 \cdot p \cdot \log(n \cdot p) + n \cdot p^2)$. In addition the latest version of SG will be improved by removing the hill-climbing completely from the framework and caching the fitness of individuals in the population. This improved version of SG will be referred to as SG v1.1. SG v1.1 will be able to solve 2D Spin Glasses and NK-landscapes effectively and compete with the state-of-the-art on problems with overlapping sub-functions. It will be shown that GI is able to reduce Trap to a Counting Ones problem.

1.1 Overview of Thesis

In Chapter 2 we will give an overview of the benchmark problems and related algorithms. Chapter 3 will be devoted to SG. In this chapter all variants of SG will be introduced, a bound for several implementations of the model-building algorithm GI will be explored and the relation between other model-building techniques will be discussed (including LTGA and Krimp). In Chapter 4 we will test all known variants and versions of SG on several test problems like Trap functions, 2D Spin Glasses and NK-landscapes.

Chapter 2

Background

In this chapter we first take a look at laboratory benchmark problems to test BBO Algorithms in Section 2.1. In Section 2.2 we give a brief overview of the Linkage Tree Genetic Algorithm (LTGA) and variants.

2.1 Benchmark Problems

Since the introduction of Genetic Algorithms (GAs) researchers have been looking for well-suited problems to test GAs. To show the limits of GAs, Goldberg introduced the notion of deception in 1987 [9]. Deceptive functions or Trap functions are problems where the low order Building Blocks (BBs) generally lead away from the global optimum. The well-known example is a concatenated Trap function (Trap). The problem uses a bit-string representation and is composed of multiple concatenated sub-functions. Each sub-function counts the number of 0s in a bit-string and gives an output linear to the number if 0s in the bit-string, but the output is always smaller than the global optimum composed of only 1s. A local search method is generally led away from the global optimum because flipping a 1 to a 0 will generally result in an increase of the fitness and flipping a 0 to 1 will only constitute to an increase if it is the last 0 in the bit-string. A example of a Trap function is shown in Figure 2.1. There have been multiple deceptive functions proposed. The most important being (concatenated) Trap Functions (Trap) and hierarchical Trap Functions (hTrap). Each problem considered in this work represents solutions as binary strings.

Trap Functions

A (concatenated) Trap Function (Trap) specifies m conjoined Trap functions of length k [9]. The bits of an individual Trap functions are concatenated into a

large bit-string for Trap with problem size $n = k \cdot m$. Each Trap function τ is a function that has two optima; the global optimum is composed of 1s and the local optimum is composed of 0s. The fitness of the global optimum is referred to as f_{high} and the fitness of the local optimum is referred to as f_{low}. One individual Trap function is plotted in Figure 2.1 with $f_{high} > f_{low}$. Let u be the number of 1s in the Trap function then we have for the fitness f of a solution T composed of blocks $\tau \in T$ each containing containing u_τ ones:

$$f(\tau) = \begin{cases} f_{high} & \text{if } u_\tau \tau = k \\ \frac{k-u_\tau-1}{k-1} \cdot f_{low} & \text{otherwise} \end{cases} \tag{2.1}$$

$$f(T) = \Sigma_{\tau \in T} f(\tau) \tag{2.2}$$

The global optimum of Trap is a bit-string of all 1s with fitness $f_{high} \cdot m$. In this work we consider Trap functions of various sizes, but always with the same fitness values for the local optima of each Trap function. In this work we parameterize each Trap with $f_{high} = 1$ and $f_{low} = 0.9$.

Hierarchical if-and-Only-if

Hierarchical if-and-only-if (hIFF) problem was introduced by Watson et al. [10]. The structure of this problem is a balanced binary tree, in which each of the leaves is associated with one the bits and takes the value of that bit. Fitness is defined as the sum over the fitness of all nodes in the binary tree. The leaf nodes contribute 1 to the overall fitness independent of the value of the bit. Every internal node x in the tree contributes $2^{height(x)}$ to the overall fitness, if both children have value 1 or both have value 0. A internal node has the value 1 (or 0 respectively) if both its children have the value 1 (or 0 respectively) and NIL otherwise. The global optimum is a string composed all 1s or all 0s.

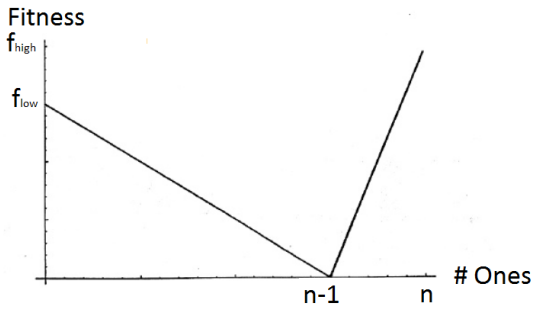

Figure 2.1: Example of a Trap function τ of n bits.

Hierarchical XOR

The hIFF problem has an optimum composed of only 1s or only 0s. This property may bias the test results towards methods that are biased to replicate particular values (e.g. all 1s or all 0s). To prevent exploitation of this particular property the hierarchical XOR (hXOR) problem was proposed by Watson et al. [11]. In hXOR the global optimum is an alternating sequence of 1s and 0s. The problem hXOR is similarly defined to hIFF, but instead of checking for equality hXOR checks for inequality [11]:

Hierarchical Trap Function

The hierarchical Trap Function (hTrap) is similar to a Trap, but uses a balanced k-ary tree as underlying structure to create hierarchy in the problem [12]. Each bit is associated with one leaf and each leaf with one bit. The tree is required to be balanced, therefore hTrap is defined in terms of height h and arity k of the tree. The number of bits n is given by $n = h^k$. Each internal node contributes to the fitness, but leafs do not contribute to the overall fitness. The fitness of each internal node is computed by a Trap function with $f_{low} = f_{high} = k^{h_{node}}$. The Trap function counts the number of 1s sent up by the nodes children. If any of the children fails to sent a 1 or a 0 up the tree then the fitness contribution of that node is 0. Internal nodes sent up a 1 or 0 up the tree if all children sent up a 1 or 0 respectively and NIL otherwise. Leafs will sent the bit value of the associated bit up the tree. The function for the root node is different. Similarly to internal nodes it is a Trap function, but now with $f_{low} = 0.9 \cdot k^{h_{root}}$ and $f_{high} = 1 \cdot k^{h_{root}}$. The hTrap problem biases the search to a solution of all 0s on each level except for the root, while the global optimum is a string of all 1s. In this work we always parameterize hTrap problems with $k = 3$.

NK-landscapes

NK-landscapes were first introduced by Kaufman [13]. In this work we will only consider Nearest Neighbour NK-landscapes. The main difference is that in Nearest Neighbour NK-landscapes the bits overlap in an ordered way instead of randomly. The fitness function is built from $m = n - k$ sub-functions that each use k bits, where $k - 1$ is the size of the neighbourhood of a bit. Fitness is calculated as the sum of fitness value's over all m functions. Each sub-function m_i has its own lookup table $\Omega(i)$ that contains a fitness value for each possible configuration of k bits. Sub-function m_i uses the k bits located on the bit-string in the interval $[i \cdot s, i \cdot s + k - 1]$, where s is the step-size. The fitness of a solution T on a Nearest Neighbour NK-landscape is then formally defined as:

$$f(T) = \Sigma_{i=0}^{m-1} \Omega_i(x_{i \cdot s}, .., x_{i \cdot s+k-1}) \tag{2.3}$$

The global optimum for Nearest Neighbour NK-landscapes depends on the random values for the sub-functions and can be found using Pelikans branch and bound solver in polynomial time [14]. In this work we only consider Nearest Neighbour variant of NK-landscapes and refer to this as NK-landscapes. NK-landscapes are always parameterised with $k = 5, s = 1$. The values in the look-up tables are uniformly distributed between 0 and 1.

2D Spin Glasses

The 2D Spin Glass problem originates from solid state physics. In this problem we have a metal, often this is Copper (Cu) or Gold (Au), and there is a small amount of impurities usually about 1%. The physicists are interested in calculating the ground energy state of the system (i.e. the lowest possible energy state). Each atom has a magnetic spin that can either be +1 or -1, we say up or down respectively. If the spins of two neighbouring atoms are either both up or both down, then we say that spins of the atoms line up. Neighbouring atoms have a coupling constant $J \neq 0$. If $J < 0$ then it is beneficial with respect to the lowest energy ground state that the spins line up, otherwise it is beneficial that the spins don't line up. The coupling constant J is 0 for non-neighbouring atoms. The energy of the system is calculated using the Hamiltonian H, which is defined as [15]:

$$H = -\Sigma_{i=0}^{n-1}\Sigma_{j=i+1}^{n-1}\ S_i \cdot J_{i,j} \cdot S_j \tag{2.4}$$

There exist many variants for the Spin Glass problem, but here we consider the 2D variant where the atoms are located on a torus and each atom has two neighbours in each dimension. Similarly to the NK-landscapes used in this work, the size of neighbourhood of each bit is 4. The coupling constant for neighbours has equal probability for both +1 and -1, but is 0 for non-neighbouring atoms. For the 2D variant of Spin Glasses polynomial time algorithms exist, but the 3D variant is NP-hard [16]. In this work we will only consider the 2D case, this subset of problems is known as 2D±J Spin Glasses [15].

2.2 Gene-pool Optimal Mixing Evolutionary Algorithms

Another approach of solving decomposable problems is to learn the linkage between the variables and use this information to perform recombination. The Linkage Tree Genetic Algorithm (LTGA) [2] does precisely this in a hierarchical fashion. Gene-pool Optimal Mixing Evolutionary Algorithms (GOMEA) is a generalization of LTGA. GOMEA is a class of evolutionary algorithms that use intermediate evaluations to see if a mixing operation was successful, this is named Optimal Mixing (OM) [3]. During recombination in GOMEA a family

of subsets (FOS) $\mathbb{F}$ is used as a guide. The FOS is a subset of the power set $\mathbb{P}$ of the set of variables $\mathbb{F} \subseteq \mathbb{P}$. Each of these sets of variables in $\mathbb{F}$ can be referred to as a mask, thus a mask is a subset of the set of variables. When GOMEA searches for new solutions. GOMEA will create a BB space for each individual and search it. To construct a BB space GOMEA will randomly select a parent for each mask in the FOS. From this parent and mask GOMEA creates a BB with a value from the parent for every variable in the mask. GOMEA will then perform a neighbourhood search in the BB space defined by these BBs. The algorithm is outlined in Algorithm 1. Mixing in GOMEA is referred to as Optimal Mixing (OM) because all masks in the FOS will be tested, using intermediary evaluations to look for an optimal solution in the BB space defined by the selected parents and current FOS [17]. The elements in the FOS are ordered (e.g. LTGA traverses the masks in reversed order or merging). Unlike in GAs, here multiple donor solutions are randomly selected from the population.

$$\mathbb{F} \subset \mathbb{P}(S) \tag{2.5}$$

Algorithm 1 Gene-pool Optimal Mixing Evolutionary Algorithm (GOMEA).

```
    ℙ ←Initial population
    while Termination criteria not met do
      if 𝔽 not predetermined then
        𝔽 ← BuildFos(ℙ)
 5:   end if
      for all i ∈ {0, 1, ..., n − 1} do
        b ← o ← ℙ_i
        fitness[b] ← fitness[o] ← fitness[ℙ_i]
        for all j ∈ {0, 1, ..., |𝔽| − 1} do
10:       p ← Random({ℙ_0, ..., ℙ_{n−1}})
          o_{F^i} ← p_{F^i}
          if o_{F^i} ≠ b_{f^i} then
            EvaluateFitness(o)
            if fitness[o] > fitness[b] then
15:           b_{f^i} ← o_{f^i}
              fitness[b] ← fitness[o]
            else
              o_{f^i} ← b_{f^i}
              fitness[o] ← fitness[b]
20:         end if
          end if
        end for
        𝕆_i ← o
      end for
25:   ℙ ← TournamentSelection(𝕆, n, 2)
    end while
```

The FOS can be build in various ways. For example, when a Linkage Tree (LT) is constructed the algorithm is referred to as LTGA. The LT is the result of a hierarchical clustering of the variables. The LT is constructed based on the Variation of Information (VI). VI is a distance measure that measures the distance between 2 sets of variations and is based on Mutual Information (MI). MI is a measure of 2 variables dependence. VI is defined as the difference between the joint entropy H and MI. The joint entropy H is a measure of how much uncertainty associated with a random variable or set of random variables. Often instead of VI the scaled VI is used, which is scaled by entropy because VI is biased towards the number of variables in the subset.

$$H(X_k) = -\Sigma_i p_i(X_k) \log p_i(X_k) \tag{2.6}$$

$$MI(X_1, \ldots, X_l) = \Sigma_{k=1}^{l} H(X_k) - H(X_1, \ldots, X_l) \tag{2.7}$$

$$\textit{scaled } VI(X_1, X_2) = 1 - \frac{MI(X_1, X_2)}{H(X_1, X_2)} \tag{2.8}$$

During each iteration of LTGA the Linkage Tree is built from the population. First, individual clusters for each variable are created. Suppose VI is used as a information measure, then in each iteration the two clusters with the smallest VI are merged and VI is updated for the remaining clusters. This process is repeated until two clusters remain. When the algorithm terminates the clusters encountered are referred to as masks. The LT-FOS is precisely the set of masks in reversed order of merging (including the n clusters for individual variables) and LTGA uses this FOS to perform recombination.

Unfortunately, using VI as defined in Equation 2.8 can cause a large computational burden especially if we are calculating VI for large subsets of variables. A more efficient method for computing the LT is to use the Unweighted Pair Group Method with Arithmetic mean (UPGMA), shown in Equation 2.9. Using this method VI is no longer required to be calculated between large subsets of variables but only pairwise, thereby reducing the complexity of the model-building algorithm from $O(n^3 \cdot p)$ to $O(n^2 \cdot p)$ [18].

$$VI(X_1, X_2) = \frac{1}{|X_1||X_2|} \Sigma_{x_i \in X_1} \Sigma_{x_j \in X_2} H(x_i, x_j) - I(x_i, x_j) \tag{2.9}$$

The GOMEA algorithm, shown in Algorithm 1, starts off by creating a population composed of random bit-strings. A possible modification is to use a simple bit-flip hill-climber to initiate the population [19]. After the population has been initiated, the first iteration of GOMEA starts. First the FOS is build from the population, then for each individual in the population all masks in the FOS are traversed. For each mask a donor is randomly selected from the population and recombination is performed between the candidate solution and

the donor. Recombination is performed by moving the values for each of the variables in the mask to a copy of the candidate. Each time this leads to a improvement, the candidate solution is updated and the search is continued with the updated candidate solution. Once the FOS is fully traversed, the best solution encountered is copied to the next population. If no improvements have been made, then the original individual is copied instead. GOMEA performs a neighbourhood search guided by the FOS.

When solving difficult hierarchical problems Goldberg et al. have identified three keys to successfully solve difficult hierarchical problems; hierarchical decomposition, chunking and preservation of candidate solutions [12]. In LTGA problem decomposition is achieved by the hierarchical variable decomposition in the LT. LTGA does not require additional chunking of the problem space, because the LT is the result agglomerative clustering. The LT is already a very compact representation that is able to chunk pieces of the solution using the masks from the FOS. This is in sharp contrast to other Evolutionary Algorithms (including hierarchical Bayesian Optimization Algorithm (hBOA)). The hBOA algorithm builds a dependency graph over all parameters [20]. The number parameters describing the probabilities in this network may grow exponentially, for this reason hBOA needs to limit the number of parents of nodes in the dependency graph and requires local structures to be able to efficiently chunk partial solutions. Maintaining a diverse population of candidate solutions is achieved in LTGA with tournament selection. Tournament selection offers a scheme in which weak individuals still have a chance to get into the population for the next iteration while still biasing towards good solutions.

GOMEA can be seen as an hybrid between GAs and Estimation of Distribution Algorithms (EDAs) [2]. EDAs build a probabilistic model of the population to capture the global structure of the problem, whereas some good solutions may still contain information that is not captured in the global probability model. The recombinative properties of GAs might be better suited to preserve this information. LTGA combines these properties by performing recombination guided by the linkage information from the LT that captures the global structure of the problem.

Since the introduction of GOMEA and LTGA, multiple variants of linkage learning for GOMEA have been proposed. What follows for the remainder of this section is a brief overview of the recent variants of GOMEA.

In GOMEA tournament selection sometimes gives too much selection pressure than is needed and diversity is reduced faster than is required. For this reason Forced Improvement (FI) was proposed by Bosman et al. [17]. Compared to GOMEA, Gene-pool Optimal Mixing with Forced Improvements (GOM-FI) applies two modifications. The tournament selection is no longer used to select individuals for the next generation but is used to select individuals for the model. In addition, the variation operator is changed. If it does not find an improvement in the current OM then it will try to perform OM again, but now

with the best found solution as a donor. If the population is drawn to multiple basins of attraction this will force the population to go in the direction of one of these basins, enforcing convergence. These adaptations change the selection pressure such that selection pressure is not increased continuously but only if a solution could not be improved anyway. In addition, a selection procedure to select individuals for model-building has been shown to be beneficial [21].

One drawback of LT is that in LT there are limited options to represent overlapping subsets. Suppose we have four variables and the first three and the last three are linked. Then in LT there is no way to represent this as groups of linked variables, because in LT the overlapping subsets are the result of hierarchical clustering. A possible way to compute overlapping subsets of linked variables is to consider linkage neighbours from the perspective each variable. This approach is named Linkage Neighbours Family Of Subsets (LN-FOS). Using the LN-FOS with FI-GOMEA is called LNGA, but can also be referred to as LN-GOMEA. To learn a LN-FOS, a test to see if two variables are truly linked is required. A common approach is to use statistical hypothesis testing, often the likelihood test statistic is used for this. Recent results show that using the likelihood-test is equivalent to using MI as a test statistic [17]. Both these statistics are measures of how dependent two variables are, but for both measures a threshold value is needed to truly test for dependence. Different approaches can be used; a fixed threshold, randomly selecting a threshold for each variable and Adaptive threshold selection. Experiments show that using none of the approaches to test for independence while computing the LN-FOS performs as good as LTGA where the LT-FOS is used for GOMEA [17].

Multi-scale Linkage Neighbours (MLN) FOS extends LN by allowing partially overlapping subsets. Multiple sets are computed for each variable, these are subsets of each other and can be modelled as inclusive subsets. Each of these subsets is from the perspective of a single variable. To learn a MLN-FOS different approaches can be used, but it all boils down to ordering all the other dependent variables in order of decreasing dependence using VI and selecting prefix's. One of these approaches is bucket-exponential decomposition. A bucket sort algorithm sorts the variables in $\sqrt{n}$ buckets. The contents of the first bucket are always added to the result and keep adding the next bucket while it contains at least twice the number of variables as the previous bucket. Experiments show that using this bucket-exponential decomposition is the most efficient among the approaches that were tested. These experiments also show that MLNGA is more efficient than LNGA. This is explained by the fact that MLNGA is able to express linkages at different levels [17].

In the latest variant of GOMEA, Linkage Trees and Neighbours Genetic Algorithm (LTNGA) [4], different linkage models are combined to construct a model that combines the strengths of the different linkage models. In this variant the LT-FOS and the MLN-FOS are combined into one FOS and filtered. Both FOS contain linkage hierarchies, such that $F_i \subset F_j$. In this Linkage Trees

and Neighbours Family Of Subsets (LTN-FOS) it is possible that we also have $F_k \subset F_j$ in the FOS. To see if parent F_j can be removed, the algorithm tests whether F_i and F_k are truly independent. If this is the case then F_j should not tried any-more because it is composed of two strongly linked sets of variables. To consider removing child F_i from the FOS, the algorithm tests the Linkage Strength (LS) of F_j and F_i. If the variables in F_j are more strongly linked than in F_i, then this is an indication that combinations made with the smaller building blocks contained by F_i should not be tried any-more. LS is defined in Equation 2.10.

$$LS(F) = \begin{cases} \frac{1}{\frac{1}{2}|F|(|F|-1)} \Sigma_{i=0}^{|F|-1} \Sigma_{j=i+1}^{|F|-1} MI(F_i, F_j) & \text{if } |F| > 1 \\ H(F_0) = MI(F_0, F_0) & \text{if } otherwise \end{cases} \tag{2.10}$$

The child filtering can be applied to both the LT and the MLN. The parent filtering will only be effective in LT, because MLN already uses a form of parent filtering. Results for LTGA, LNGA and MLNGA are shown in the next section on various test problems (including Trap and NK-landscapes).

Scaling of GOMEA Related Algorithms

The most recent results were published in [4]. Here the scalability of LTGA, MLNGA was tested on several testing problems, including the NP-complete MAXCUT.

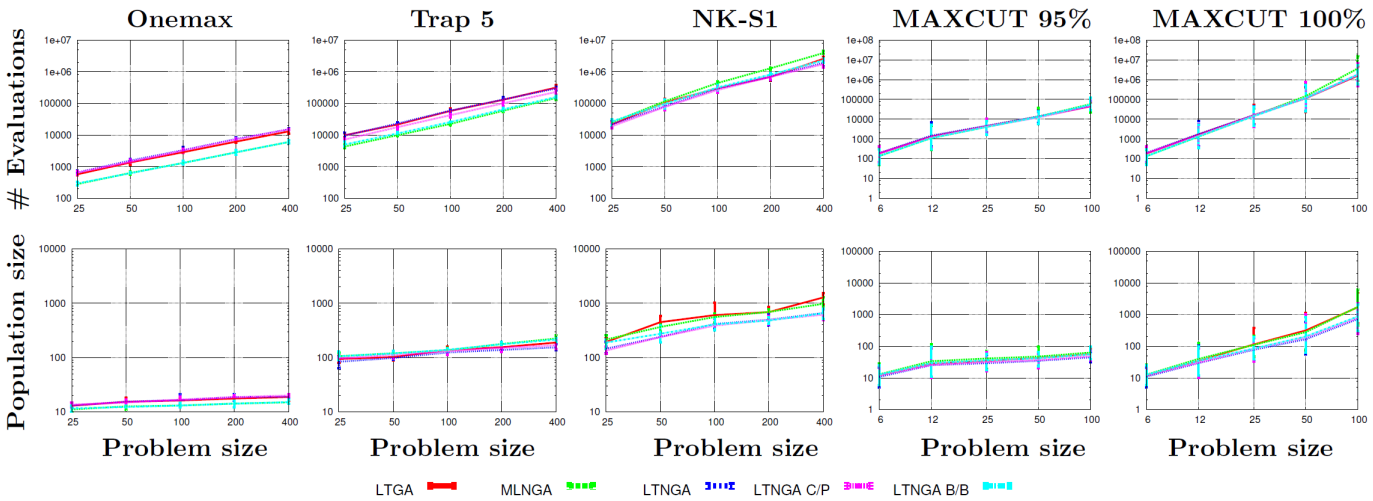

Figure 2.2: MLNGA on Onemax, Trap, NK, and MAXCUT. This image was originally published by Bosman et al. in [4].

Chapter 3

Multi-Scale Search: Schema Grammar

Multi-Scale Search (MSS) algorithms are population-based Evolutionary Algorithms (EAs) [22]. They have the distinctive property to hierarchically decompose the problem and search for new solutions using chunks of solutions. Chunks are incomplete solutions and can be referred to as either chunks, schemata, variational units or Building Blocks (BBs). MSS algorithms redefine their model in each iteration to search at a higher scale of the problem using larger chunks. Formally, algorithms belonging to the MSS class are defined by the following four characteristics:

Evolutionary Search Algorithms belonging to the MSS class use a special form of local search. This local search process doesn't just flip bits, it operates in a search neighbourhood defined by BBs. Often the initial search comes down to simple bit flip hill-climbing, because only variational units specifying a single variable are used. This special form of local search can be referred to as a unit-exploiting search or neighbourhood search. The unit evolutionary search doesn't have to be a local search specifically, it can be any method as long as it searches the BB search space. Similarly to OM in GOMEA, the neighbourhood search uses intermediary evaluations during mixing to select the best solution encountered. An important difference is that it does not necessarily use a parent directly as donor.

Redefining variational units During the unit exploiting search, variational units of the current scale of the problem are used to find high fitness combinations of the next scale. In each iteration a new model is computed from these new results to search at a higher scale in the next iteration.

Separation of time-scales The processes evolutionary search and redefining variational units occur constantly in MSS algorithms. These processes

should be separated enough such that the unit exploiting search has returned enough results to redefine units for a higher scale of the problem [23]. Usually this is achieved by completely separating these processes by performing them sequentially. First, a neighbourhood search is performed to find new solutions. Then the neighbourhood search stops and the MSS algorithm redefines the variational units based on these new results. The neighbourhood search process starts again, but now with new variational units of a higher scale. This process is repeated until either the computational budget has been reached or global optimum has been found.

Diverse Sampling Like other Evolutionary Algorithms (EA), MSS algorithms require a diverse population to build a model. To achieve this, the unit exploiting search must be applied repeatedly and from different starting conditions.

In short, MSS is a class of population-based EAs. This class is loosely defined with the following four properties; Evolutionary Search, redefining variational units, separation of time-scales and diverse sampling. These algorithms look for small chunks of partial solutions and use these chunks to search the problem-space. In each iteration these chunks are redefined and the problem space is further explored using these chunks in a unit-exploiting search.

3.1 Schema Grammar

In this section Schema Grammar (SG) is described, based on the work of Cox et al. [5] [6] [7]. SG was first introduced by Chris R. Cox in July 2014. In this work a Context-free Formal Grammar (CFG) is introduced as Schema Grammar (SG). A CFG is a simple and precise mechanism for describing natural languages that are built from smaller blocks and captures this block structure in a natural way. SG is a framework that allows the population to be modelled and compressed using BBs. The resulting BBs can be used in an evolutionary search to find new high fitness combinations.

In SG solutions are represented using the messy Genetic Algorithm Encoding (mGAE) [24]. In mGAE solutions are represented as sets of symbols. These symbols can either represent a specific value for a variable, which is referred to as a Terminal Symbol (TS), or a Non-terminal Symbol (NS). A TS is a variable-value tuple and can be referred to as an allele. To express a TS in mGAE we write $\langle k|X_k\rangle$ for variable k with value X_k. A NS represents either complete solutions or partial solutions (schemata), they are simply expressed as the associated symbol, for example s_0 or s_1. Suppose we want to encode the solution ($g = 01100$) then in mGAE this is expressed using only TS as $g = \{\langle 0|0\rangle, \langle 1|1\rangle, \langle 2|1\rangle, \langle 3|0\rangle, \langle 4|0\rangle\}$, where the order of the tuples in the set is irrelevant.

A NS represents a set of non-overlapping symbols, but the set can contain both NS and TS. Let S be the set of NS that do not represent individuals in the solution, but are partial solutions (i.e. they can not contain a value for every variable when they are recursively expanded). When the symbols $s \in S$ are (recursively) expanded, the result is a set of BBs. The set S can be thought of as a set of schemata. It is precisely this set that is used as variational units in Schema Search. Schema Search is a unit exploiting search algorithm used in SG search algorithms, more on this in Section 3.1.4.

The NS $g \in G$ represent individuals in the population. After recursively expanding a NS $g \in G$, the resulting set of TS is a set that specifies the value for every variable exactly once. The fact that individuals in the population are required to be fully specified is in sharp contrast to regular mGAE, where under-specification and over-specification is allowed (the NS $s \in S$ allow for under-specification). Let V be the complete set of NS, defined in Equation 3.1, and let Σ be the set of all TS that are currently in the grammar, defined in Equation 3.2, then there exists a production rule $r_v \in R$ with right-hand side β_v for every NS $v \in V$:

$$V = G \cup S \tag{3.1}$$

$$\Sigma = \{\langle k|X_k\rangle | v \in V, \langle k|X_k\rangle \in \beta_v\} \subseteq \{\langle k|X_k\rangle | k \in \{0...n-1\}, X_k \in \{0,1\}\} \tag{3.2}$$

$$r_v \in R :: V \rightarrow \{S \cup \Sigma\}^* \tag{3.3}$$

$$\beta_v \subseteq S \cup \Sigma \tag{3.4}$$

The complete set of alleles that are contained in NS s_0 can be found by recursively expanding s_0. To recursively expand s_0, first the symbol s_0 is replaced by each symbol in the right side β_{s_0} of rule r_{s_0}. Each NS $s \in S$ that is still present is then recursively expanded by replacing it with the right hand side β_s. This process is repeated until no NS occur anywhere in the result any-more. No cycles may occur anywhere in the grammar, therefore the recursive expansion of any symbol is deterministic.

There exists a trivial and uncompressed instance of the grammar for any population of solutions. Namely the instance where there are no additional NS ($S = \emptyset$), we have a NS in G for every individual g_i in the population and each individual is initiated with a variable-value tuple for every variable. For example, in Table 3.1 we can see an uncompressed trivial instance of the grammar. The population is composed of three solutions ($g_0 = 01100$, $g_1 = 10100$, $g_2 = 00000$).

Table 3.1: Example of an uncompressed instance of the grammar.

SG set	**Value of SG set**		
V	$\{g_0, g_1, g_2\}$		
G	$\{g_0, g_1, g_2\}$		
S	$\emptyset$		
Σ	$\{\langle 0\|0\rangle, \langle 0\|1\rangle, \langle 1\|0\rangle, \langle 1\|1\rangle, \langle 2\|0\rangle, \langle 2\|1\rangle, \langle 3\|0\rangle, \langle 4\|0\rangle\}$		
R	r_0	$g_0 \rightarrow \{\langle 0\|0\rangle, \langle 1\|1\rangle, \langle 2\|1\rangle, \langle 3\|0\rangle, \langle 4\|0\rangle\}$	0 1 1 0 0
	r_1	$g_1 \rightarrow \{\langle 0\|1\rangle, \langle 1\|0\rangle, \langle 2\|1\rangle, \langle 3\|0\rangle, \langle 4\|0\rangle\}$	1 0 1 0 0
	r_2	$g_2 \rightarrow \{\langle 0\|0\rangle, \langle 1\|0\rangle, \langle 2\|0\rangle, \langle 3\|0\rangle, \langle 4\|0\rangle\}$	0 0 0 0 0

There are various ways to create and compress an instance of the grammar. In the next section, we will describe an algorithm called Grammar Inference (GI). GI creates a compressed instance based on the frequency of the co-occurrence of symbols. The result is a loss-less representation of the original population. A possible instance of the grammar after compression is shown in Table 3.2.

Typically compressed instances are created by adding rewriting rules for patterns in the data. Patterns are represented by BBs. By creating a new NS and an associated rule for each pattern the data is compressed into a factorised representation of the original data. Moreover, the recursive properties of production rules allow schemata to be modelled using multiple levels of subschemata, which leads to a compact and hierarchically decomposed problem representation. Any grammar compression is loss-less and does not contain any loops and each rule is associated with a single NS, therefore the expansion of any symbol is deterministic. GI was proposed to create compressed instances of the grammar based on correlations in the data [5], this algorithm will be discussed in Section 3.1.2.

Table 3.2: Example of an (compressed) instance of the grammar.

SG set	**Value of SG set**		
V	$\{g_0, g_1, g_2, s_0, s_1\}$		
G	$\{g_0, g_1, g_2\}$		
S	$\{s_0, s_1\}$		
Σ	$\{\langle 0\|0\rangle, \langle 0\|1\rangle, \langle 1\|0\rangle, \langle 1\|1\rangle, \langle 2\|0\rangle, \langle 2\|1\rangle, \langle 3\|0\rangle, \langle 4\|0\rangle\}$		
R	r_0	$g_0 \rightarrow \{\langle 0\|0\rangle, \langle 1\|1\rangle, s_1\}$	0 1 1 0 0
	r_1	$g_1 \rightarrow \{\langle 0\|1\rangle, \langle 1\|0\rangle, s_1\}$	1 0 1 0 0
	r_2	$g_2 \rightarrow \{\langle 0\|0\rangle, \langle 1\|0\rangle, \langle 2\|0\rangle, s_0\}$	0 0 0 0 0
	r_3	$s_0 \rightarrow \{\langle 3\|0\rangle, \langle 4\|0\rangle\}$	### 0 0
	r_4	$s_1 \rightarrow \{\langle 2\|1\rangle, s_0\}$	## 1 0 0

3.1.1 Formal Definition

Let R be the total set of all rules, then SG is formally defined by the a 5-tuple:

$$\mathbb{G}_{schema} = (V, G, S, \Sigma, R) \tag{3.5}$$

V is a non-empty set of all NS, with $V = G \cup S$.

G is a non-empty set of NS representing the individuals in the population. The recursive expansion of any symbol in G specifies a value for every variable exactly once.

S is a set of NS representing schemata, which may be empty in an (uncompressed) instance of the grammar.

Σ is the set of TS present anywhere in the grammar. This set is a subset of the set of all possible complete set of alleles in an n-dimensional binary problem space encoded as $\langle$position|value$\rangle$ tuples or more formally:
$\Sigma = \{\langle k|X_k\rangle | v \in V, \langle k|X_k\rangle \in \beta_v\} \subseteq \{\langle k|X_k\rangle | k \in \{0...n-1\}, Xk \in \{0,1\}\}$.

R is a set of production rules that relate each NS to the expansion of an unordered finite set of symbols, such that $R :: V \rightarrow \{V \cup \Sigma\}^*$.

Pre-terminal Symbols

There is one special set of NS, this set is called the Pre-terminal symbols P. We will need this set later when we describe one of the variants of Schema Grammar, named SG-Trap. This set includes every NS s_0 from S that includes a TS directly in its expansion:

$$P = \{s \in S | \beta_s \cap \Sigma \neq \emptyset\} \tag{3.6}$$

3.1.2 Grammar Inference

Grammar Inference (GI) is an off-line loss-less compression algorithm that compresses the population into a compressed instance of SG. The algorithm is based on two existing compression algorithms; RE-PAIR (an off-line algorithm from Larsson and Moffat [25]) and SEQUITUR (an on-line algorithm from Nevill-Manning and Witten [26]). The main principle of both these algorithms is to infer dependency from co-occurrence. In contrast to these algorithms, co-occurrence for GI is defined as two symbols being members of the same set. This definition of co-occurrence is order independent as opposed to two symbols appearing next to each other in a string. For example, if we have the production rule $s \rightarrow \{a, b, c\}$, then in RE-PAIR and SEQUITUR we have two co-occurrences; $\{a, b\}$ and $\{b, c\}$. In SG we would have three co-occurrences; $\{a, b\}$, $\{b, c\}$ and $\{a, c\}$. This is the reason that SG is free from any positional

Algorithm 2 Grammar Inference (GI), the off-line compression algorithm for inferring an instance of SG. This is a slightly modified version of the original pseudo-code. The original pseudo-code was published by Cox et al. in [5].

Input: Population $\mathbb{P} = \{P_1, P_2, \ldots, P_p\}$
Output: grammar instance $\mathbb{G} = \{V, G, S, \Sigma, R\}$

```
    G ← V ← R ← Σ ← ∅
    for all P ∈ ℙ do
      for all α ∈ P do
        Σ ← Σ ∪ α
 5:   end for
      g ← NewSymbol()
      G ← G ∪ g
      R ← R ∪ (g → P)
    end for
10: ℂ ← ∅ {Please note, ℂ is a bag}
    for all (α → β) ∈ R do
      ℂ ← ℂ ⊎ Co-occur(β)
    end for
    while F(ℂ) ≠ ∅ do
15:   s ← NewSymbol()
      S ← S ∪ s
      R ← R ∪ (s → F(ℂ))
      M ← Find(G, F(ℂ))
      for all (α → β) ∈ M do
20:     ℂ ← ℂ \ {{A, B} ∈ Co-occur(β) : A ∈ F(ℂ) ∨ B ∈ F(ℂ)}
        β ← s ∪ {β \ F(ℂ)}
        ℂ ← ℂ ⊎ {{A, B} ∈ Co-occur(β) : s ∈ {A, B}}
      end for
    end while
25: U ← FindUnderutilised(R, S)
    for all (s, β_s, β_r) ∈ U do
      β_r = β_s ∪ {β_r \ s}
      S ← S \ s
    end for
30: return {G ∪ S, G, S, Σ, R}
```

biases. Like SEQUITUR, after all the co-occurrences have been identified and removed, the algorithm will collapse any superfluous nested hierarchies of symbol pairs. A schemata s_i is collapsed if it is only referenced once. In this case the algorithm will remove both the symbol s_i and rule r_i from the grammar and replace the only occurrence with the right-hand side β_i. Pseudo-code for this algorithm is shown in Algorithm 2.

To show how GI works a small example is discussed next. Starting from a population composed of three members (g_0 = 011000, g_1 = 101000, g_2 = 000000). GI starts off in lines 1 to 9 to create an empty instance of the grammar with the individuals encoded as rules, then GI creates a bag for all tuples of

symbols that occur anywhere on the right-hand side of rules in R in lines 10 to 13. The function Co-occur$(\beta) = \{A \in \mathcal{P}(\beta) : |A| = 2\}$ selects every pair of co-occurring tuples of symbols from every individual in the population in $O(|\beta|^2) = O(n^2)$ time. The resulting instance is shown in Table 3.3.

The while-loop in lines 14 to 24 is executed as long as there are symbol pairs co-occurring more than once. In each iteration the algorithm will compute the most frequent pair of co-occurring symbols and remove it from the grammar completely. GI replaces every occurrence of the tuple with a new symbol and adds a new production rule to R with the most frequent tuple on the right side of the production rule. The operation $F :: \mathbb{C} \to \Sigma \times \Sigma$ finds the most frequent tuple from $\mathbb{C}$, which worst time complexity depends on the data structure chosen for $\mathbb{C}$. Multiple implementations will be discussed in detail in Section 3.1.3 (including an array and a hash table). After creating the new rule, GI finds all rules that contain $F(\mathbb{C})$ in line 18. The Find operation finds all symbols $v \in V$ containing the most frequent tuple $F(\mathbb{C})$ on the right side of the production rule p_v. Finally, in lines 19 to 23 the right hand side of the rules is updated, a new symbol is added and $\mathbb{C}$ is updated. By recursively identifying and creating rules, larger schemata are formed when one of the co-occurring symbols is a member of S ($F(\mathbb{C}) \cap S \neq \emptyset$).

In the example the tuples $(\langle 3|0\rangle, \langle 4|0\rangle), (\langle 3|0\rangle, \langle 5|0\rangle), (\langle 4|0\rangle, \langle 5|0\rangle)$ all have the largest number of occurrences. GI randomly selects the tuple $(\langle 4|0\rangle, \langle 5|0\rangle)$ and creates symbol s_0 and rule r_0 for this tuple. The resulting instance is shown in Table 3.4. The second time the algorithm finds $(\langle 3|0\rangle, s_0)$ as most frequent occurring tuple. GI creates rule r_1 for this tuple and updates the grammar. The resulting instance is shown in Table 3.5. The last time the while-loop is executed $(\langle 2|1\rangle, s_1)$ is the only frequent co-occurring pair of symbols. The resulting instance is shown in Table 3.6.

In the second phase of the algorithm the operation FindUnderutilised is executed to compute the set of superfluous symbols $U \subset S$. These symbols are superfluous because they are used exactly once. Each symbol $s \in U$ is removed from the grammar and the single occurrence s is replaced with β_s in lines 26 to 29. In the example the only NS that is referenced once is s_0. After removal, the

Table 3.3: Example of the initial situation before grammar inference.

SG set	**Value of SG set**		
V	$\{g_0, g_1, g_2\}$		
G	$\{g_0, g_1, g_2\}$		
S	$\emptyset$		
Σ	$\{\langle 0\|0\rangle, \langle 0\|1\rangle, \langle 1\|0\rangle, \langle 1\|1\rangle, \langle 2\|0\rangle, \langle 2\|1\rangle, \langle 3\|0\rangle, \langle 4\|0\rangle\ \langle 5\|0\rangle\}$		
R	r_0	$g_0 \to \{\langle 0\|0\rangle, \langle 1\|1\rangle, \langle 2\|1\rangle, \langle 3\|0\rangle, \langle 4\|0\rangle\ \langle 5\|0\rangle\}$	0 1 1 0 0 0
	r_1	$g_1 \to \{\langle 0\|1\rangle, \langle 1\|0\rangle, \langle 2\|1\rangle, \langle 3\|0\rangle, \langle 4\|0\rangle\ \langle 5\|0\rangle\}$	1 0 1 0 0 0
	r_2	$g_2 \to \{\langle 0\|0\rangle, \langle 1\|0\rangle, \langle 2\|0\rangle, \langle 3\|0\rangle, \langle 4\|0\rangle, \langle 5\|0\rangle\}$	0 0 0 0 0 0

Table 3.4: Example after generating the first rule.

SG set	**Value of SG set**		
V	$\{g_0, g_1, g_2, s_0\}$		
G	$\{g_0, g_1, g_2\}$		
S	$\{s_0\}$		
Σ	$\{\langle 0\|0\rangle, \langle 0\|1\rangle, \langle 1\|0\rangle, \langle 1\|1\rangle, \langle 2\|0\rangle, \langle 2\|1\rangle, \langle 3\|0\rangle, \langle 4\|0\rangle, \langle 5\|0\rangle\}$		
R	r_0	$g_0 \rightarrow \{\langle 0\|0\rangle, \langle 1\|1\rangle, \langle 2\|1\rangle, \langle 3\|0\rangle, s_0\}$	0 1 1 0 0 0
	r_1	$g_1 \rightarrow \{\langle 0\|1\rangle, \langle 1\|0\rangle, \langle 2\|1\rangle, \langle 3\|0\rangle, s_0\}$	1 0 1 0 0 0
	r_2	$g_2 \rightarrow \{\langle 0\|0\rangle, \langle 1\|0\rangle, \langle 2\|0\rangle, \langle 3\|0\rangle, s_0\}$	0 0 0 0 0 0
	r_3	$s_0 \rightarrow \{\langle 4\|0\rangle, \langle 5\|0\rangle\}$	#### 0 0

Table 3.5: Example after generating the second rule.

SG set	**Value of SG set**		
V	$\{g_0, g_1, g_2, s_0, s_1\}$		
G	$\{g_0, g_1, g_2\}$		
S	$\{s_0, s_1\}$		
Σ	$\{\langle 0\|0\rangle, \langle 0\|1\rangle, \langle 1\|0\rangle, \langle 1\|1\rangle, \langle 2\|0\rangle, \langle 2\|1\rangle, \langle 3\|0\rangle, \langle 4\|0\rangle, \langle 5\|0\rangle\}$		
R	r_0	$g_0 \rightarrow \{\langle 0\|0\rangle, \langle 1\|1\rangle, \langle 2\|1\rangle, s_1\}$	0 1 1 0 0 0
	r_1	$g_1 \rightarrow \{\langle 0\|1\rangle, \langle 1\|0\rangle, \langle 2\|1\rangle, s_1\}$	1 0 1 0 0 0
	r_2	$g_2 \rightarrow \{\langle 0\|0\rangle, \langle 1\|0\rangle, \langle 2\|0\rangle, s_1\}$	0 0 0 0 0 0
	r_3	$s_0 \rightarrow \{\langle 4\|0\rangle, \langle 5\|0\rangle\}$	#### 0 0
	r_4	$s_1 \rightarrow \{\langle 3\|0\rangle, s_0\}$	### 0 0 0

Table 3.6: Example after generating the all rules.

SG set	**Value of SG set**		
V	$\{g_0, g_1, g_2, s_0, s_1, s_2\}$		
G	$\{g_0, g_1, g_2\}$		
S	$\{s_0, s_1, s_2\}$		
Σ	$\{\langle 0\|0\rangle, \langle 0\|1\rangle, \langle 1\|0\rangle, \langle 1\|1\rangle, \langle 2\|0\rangle, \langle 2\|1\rangle, \langle 3\|0\rangle, \langle 4\|0\rangle, \langle 5\|0\rangle\}$		
R	r_0	$g_0 \rightarrow \{\langle 0\|0\rangle, \langle 1\|1\rangle, s_2\}$	0 1 1 0 0 0
	r_1	$g_1 \rightarrow \{\langle 0\|1\rangle, \langle 1\|0\rangle, s_2\}$	1 0 1 0 0 0
	r_2	$g_2 \rightarrow \{\langle 0\|0\rangle, \langle 1\|0\rangle, \langle 2\|0\rangle, s_1\}$	0 0 0 0 0 0
	r_3	$s_0 \rightarrow \{\langle 4\|0\rangle, \langle 5\|0\rangle\}$	#### 0 0
	r_4	$s_1 \rightarrow \{\langle 3\|0\rangle, s_0\}$	### 0 0 0
	r_5	$s_2 \rightarrow \{\langle 2\|1\rangle, s_1\}$	## 1 0 0 0

Table 3.7: Final instance of the example after removing all superfluous rules.

SG set	**Value of SG set**		
V	$\{g_0, g_1, g_2, s_1, s_2\}$		
G	$\{g_0, g_1, g_2\}$		
S	$\{s_1, s_2\}$		
Σ	$\{\langle 0\|0\rangle, \langle 0\|1\rangle, \langle 1\|0\rangle, \langle 1\|1\rangle, \langle 2\|0\rangle, \langle 2\|1\rangle, \langle 3\|0\rangle, \langle 4\|0\rangle, \langle 5\|0\rangle\}$		
R	r_0	$g_0 \to \{\langle 0\|0\rangle, \langle 1\|1\rangle, s_2\}$	0 1 1 0 0 0
	r_1	$g_1 \to \{\langle 0\|1\rangle, \langle 1\|0\rangle, s_2\}$	1 0 1 0 0 0
	r_2	$g_2 \to \{\langle 0\|0\rangle, \langle 1\|0\rangle, \langle 2\|0\rangle, s_1\}$	0 0 0 0 0 0
	r_4	$s_1 \to \{\langle 3\|0\rangle, \langle 4\|0\rangle, \langle 5\|0\rangle\}$	### 0 0 0
	r_5	$s_2 \to \{\langle 2\|1\rangle, s_1\}$	## 1 0 0 0

final result is shown in Table 3.7.

In the example the compression returned two schemata; $s_1 = (***000)$ and $s_2 = (**1000)$, both these schemata can be used as variational units in an unit exploiting search.

Schema Hierarchy

After GI has compressed the population into an instance of SG, the resulting individuals are expressed as a schema hierarchy. This can be attributed to the recursive properties of the grammar. The schema hierarchy is a hierarchical problem decomposition on a variable-value level. This is in sharp contrast to the LT in LTGA or the FOS in GOMEA in general because the FOS is a problem decomposition on a variable level. In Figure 3.1 the schema hierarchy of a solution to a NK-Landscape is shown. In the image we see the solution expressed in a hierarchy of BBs. On the first row of the image we can see the solution of the individual g on a NK-landscape. In the rows below we can see the expansions of all the NS that were directly or recursively referenced in the right hand side β_g. If we look at row 2, then we see the expansion of a symbol $s_2 \in \beta_g$. The arc pointing at row 2 from 4 shows which genes were inherited from the symbol that was expanded at row 4 $s_4 \in \beta_{s_2}$, etc.

3.1.3 Complexity Grammar Inference

In this section we will explore several implementations for the data-structure of $\mathbb{C}$ in GI. This object stores all pairs of symbols and is used to compute the most frequent occurring pair of symbols with operation F. We will explore various implementations for $\mathbb{C}$ (including a flat array, a hash table and an array of fully balanced k-ary trees). In the proofs on the implementations we will only take the operation F and the update operations to the data-structure $\mathbb{C}$ into account because the remainder of the algorithm is independent of the implementation chosen for $\mathbb{C}$.

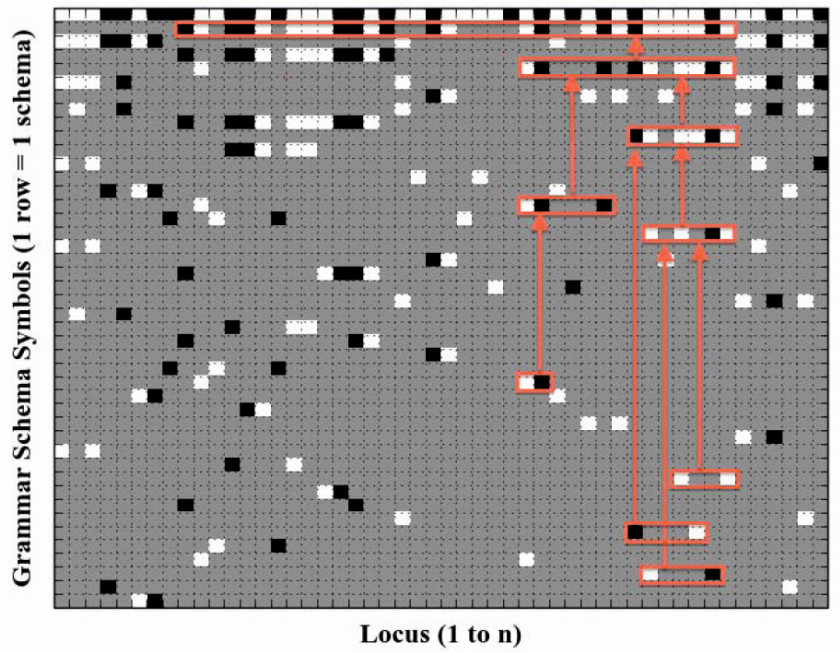

Figure 3.1: Example of a unshuffled schema hierarchy for a individual on a NK-landscape. In this image we see on the first row we see the fully specified solution. The columns represent variables, one column for one variable. Black represents a 0, White a 1 and grey an unspecified variable. When the solution was recursively expanded, each referenced schemata was added as a row below. When we follow the orange arcs we see a example of such schema hierarchy. Suppose we have an arc from row i to row j. And row i contains schemata s_i and row j schemata s_j means that s_j contains the NS s_i directly in its expansion p_{s_j}. Locus is a term from theoretical biology used to describe a single variable. This image was originally published by Cox et al. in [6].

In this work we have modified the GI algorithm. Originally, the FIND operation in line 18 looks for the most frequent co-occurring tuple in R [7], but at this point in the algorithm there is no right-hand side of any rule $r_s \in R$ for a symbol in S that contains more than 2 symbols. In addition, the symbol pairs in the right-hand side of symbols in S ($\{\beta_s | s \in S\}$ have preciously been selected as most frequent co-occurring tuple and occur only together in that particular rule. For these reasons checking all rules for symbols in S is needless and the FIND operation in line 18 is only required to check the rules for individuals in G $\{\beta_g | g \in G\}$. This optimization reduces the worst-time complexity for all FIND operations from $O(n^2 \cdot p^2)$ to $O(n \cdot p^2)$.

Theorem 1. *The number of schemata generated $|S|$ is $O(n \cdot p)$.*

Proof. Each time a rule is generated, the total number of symbols occurring on the right side of the production rules associated with individuals in the population $g \in G$ is decreased by at least two. Rules are only removed after all rules have been generated, therefore all rules not deriving from individuals will contain exactly two symbols until all superfluous rules are removed. Rules are only created for tuples that co-occur at least twice, therefore when a new rule is created with GI at least four or more characters are removed. This may be more if the tuple occurs more than twice, because for each occurrence the number of symbols in the grammar decreases by one. In the worst-case, when every solution occurs exactly twice, there are exactly $(n-1) \cdot p$ of such symbols that can be removed. Therefore, there are at most $\frac{(n-1)\cdot p}{2} = O(n \cdot p)$ rules created. □

In practice, we may expect less rules to be generated during the first phase of the algorithm. Usually the occurrences of tuples is higher than two, thereby reducing computational effort greatly. Unfortunately, we are not able to derive a tighter bound without making more assumptions about the problem.

Theorem 2. *The number of update operations to the data structure $\mathbb{C}$ in GI s $O(n^2 \cdot p)$.*

Proof. (This proof was originally suggested by Chris Cox [7]) For any given production rule the operation Co-occur(β) requires each unique pair of symbols in β to be identified, which requires $\frac{|\beta|\cdot(|\beta|+1)}{2} = O(n^2)$ tuples in each individual to be identified. Therefore, the initial structure will contain $(n^2 \cdot p)$ tuples.

The maximum number of symbol substitutions per individual is $n-1$. On each substitution the length of the RHS of the rule $|\beta|$ is decremented by 1. Therefore, the size of each rule changes from n to 2. For each substitution the update of $\mathbb{C}$ on line 20 requires $2(|\beta|-1)$ operations, $(|\beta|-1)$ for each symbol in the pair being substituted and the update of $\mathbb{C}$ on line 22 requires $|\beta|-1$ operations. So on each substitution there are $3(|\beta|-1)$ operations. Summing these operations across the series $|\beta| = (n, n-1, \ldots, 2)$ yields for each individual:

$$3(1+2+3\ldots(n-2)(n-1)) = O(n^2) \tag{3.7}$$

Yielding a total number of $O(n^2 \cdot p)$ operations to keep $\mathbb{C}$ up to date. □

Theorem 3. *Excluding the F operations and all updates to $\mathbb{C}$ in GI, the computational complexity of GI is $O(n^2 \cdot p + n \cdot p^2)$.*

Proof. The lines 1 to 9 of GI (shown in Algorithm 2) take $O(n \cdot p)$ time, because we need to consider every symbol of every individual to construct Σ. Line 10 is performed in constant time. The lines 11 to 13 take $O(n^2 \cdot p)$ time, because we need to find every $O(n^2)$ tuple in every individual.

Excluding the F operations and all updates to $\mathbb{C}$, the lines 14 to 24 are at most executed once for every rule, that is $O(n \cdot p)$ times. In each iteration the FIND operation is executed and finds all rules containing the most frequent tuple in $O(p)$. This can be performed in linear time because we only need to consider symbols representing individuals (symbols $s \in S$ are always of size two and contain tuples that can not be contained in any rule at this point in the algorithm). The update in line 21 decreases the number of symbols representing individuals by at least 1, therefore the maximal number that this operation can be performed in $O(n \cdot p)$. Summing it all up, the lines 14 to 24 (excluding the F operations and all updates to $\mathbb{C}$) take $O(n^2 \cdot p + n \cdot p^2)$ time.

For finding all superfluous rules all symbols have to be iterated, this can be performed in $O(n^2 \cdot p)$ by iterating every symbol pair of every individual. Removing them can be performed in $O(n \cdot p)$, because there are at most $O(n \cdot p)$ rules to be removed and each rule is composed of two symbols with exactly one co-occurrence. Thus, removing all superfluous rules in lines 25 to 29 can be performed in $O(n^2 \cdot p)$.

Summing it all up, all operations (excluding the F operations and all updates to $\mathbb{C}$) can be performed in $O(n^2 \cdot p + n \cdot p^2)$. □

A possible implementation for $\mathbb{C}$ is an array of integers. The array has an entry for every possible combination of variable-value tuples, so we have an entry for each element in the set $(\Sigma \cup S) \times (\Sigma \cup S)$. The integers in the array describe the number of occurrences of the tuple. To find the maximal co-occurring tuple all tuples in the hash table have to be checked. We have no way to keep track of the maximal co-occurring tuple in a hash table because the data-structure is required to handle both increments and decrements. For this reason we need to iterate all tuples in either the population or the array.

Theorem 4. *Using an array as underlying data-structure for $\mathbb{C}$ in GI, the operations F and the update operations to keep the data-structure $\mathbb{C}$ up to date take $O(n^3 \cdot p^3)$ time with space requirements $O(n^2 \cdot p^2)$.*

Proof. From Theorem 1 it follows that there are at most $O(n \cdot p)$ rules generated. Each rule is associated with a symbol, therefore we can have $2 \cdot n + \frac{n \cdot p - n}{2} = O(n \cdot p)$ symbols, which means that there can be $(2 \cdot n + \frac{n \cdot p - n}{2})^2 = O(n^2 \cdot p^2)$ different tuples. The array contains a value for every tuple, so we have $O(n^2 \cdot p^2)$ tuples in the array. Each entry in the array will be associated with the number of occurrences of a tuple. The access and update time for any number in the array is $O(1)$, so initially building the array takes $O(n^2 \cdot p)$. Each update operation is handled in constant time, from this and Theorem 2 it follows that the total time to keep the array up to date is $O(n^2 \cdot p)$.

To find the maximal element either all entries in the array or all the symbols in the grammar have to be checked. We have at most $O(n^2 \cdot p^2)$ entries in the array or $O(n^2 \cdot p)$ symbols in the grammar and we need to iterate all of

them for every rule, therefore performing all these operations takes $O(n^3 \cdot p^3)$ or $O(n^3 \cdot p^2)$ respectively. Yielding the total worst-case time complexity of $O(n^3 \cdot p^3)$ or $O(n^3 \cdot p^2)$ depending which entries are iterated with space requirements of $O(n^2 \cdot p^2)$. □

An alternative to the array implementation is a hash table. In the hash table the keys are populated by tuples and the values are the number of occurrences of that particular tuple. In contrast to the array implementation, the hash table does not have empty entries in the data-structure $\mathbb{C}$, thereby greatly reducing the space requirements.

Theorem 5. *Using an hash table as underlying data-structure for $\mathbb{C}$ in GI, the operations F and the update operations to keep the data-structure $\mathbb{C}$ up to date take $O(n^4 \cdot p^2)$ or $O^*(n^3 \cdot p^2)$ time with space requirements $O(n^2 \cdot p)$.*

Proof. The initial hash table contains $O(n^2 \cdot p)$ elements. Inserting them takes constant amortized time $O^*(1)$ and the worst-case time complexity is $O(n)$. The worst-case time complexity for building the hash table initially is $O(n^4 \cdot p^2)$ or in amortized time $O^*(n^2 \cdot p)$.

The maximal co-occurring tuple is found by iterating over all elements in the hash table in $O(n^2 \cdot p)$. This can occur as often as we can create rules so $O(n \cdot p)$ times. Iterating all elements for each rule takes then $O(n^3 \cdot p^2)$. From Theorem 2 we know that the number of update operations to the data-structure is $O(n^2 \cdot p)$. Again we can access them in constant amortized time, but linear in the worst-case. This yields $O(n^4 \cdot p^2)$ or in amortized time $O^*(n^3 \cdot p^2)$ for the hash table implementation.

Summing it all up gives a worst-case time complexity $O(n^4 \cdot p^2)$, with memory requirements of $O(n^2 \cdot p)$. □

When we consider amortized time in the proof of Theorem 5, the complexity of inserting the tuples in the hash table drops down to $O^*(n^3 \cdot p^2)$. This reduction of the computational complexity is limited by the operation to find the maximal co-occurring set of tuples. This operation needs to be performed for each rule and is required to iterate $O(n^2 \cdot p)$ tuples in the hash table each time.

In the proofs of Theorem 4 and 5 the most most expensive operation performed in GI is F (i.e. the operation to find the most frequent occurring tuple). This is an indication that if we optimize the data-structure for this operation and pre-process the data to ease this task, then we may find a data-structure that leads to a lower computational complexity for GI. To this end, we propose a data-structure that is optimized specifically for this operation. We propose to store the number of occurrences for each tuple in a balanced k-ary tree L. In addition, we also store the tuples in an array T of size p filled with fully balanced k-ary trees. A tree is fully balanced if it is balanced and all nodes on the bottom layer are located as far to the left as possible. In Figure 3.2 we see

a tree that is balanced but not fully balanced. It is not fully balanced because node 6 is not fully balanced (i.e. it can be moved to the left to be the left child of node 3, like shown in Figure 3.3).

The idea is to store the number of co-occurrences in 2 ways. Once to quickly find out the number of occurrences for a tuple and once to quickly randomly select the maximal frequent co-occuring tuple. Suppose we want to store the number of co-occurrences o for a tuple t, then the tree L contains an entry indexed by t with the count of tuple t and a pointer to a node in T. The tree in $T[o']$ will contain all tuples t' with number of co-occurrences o' and each node in T will contain a pointer to the node in L representing that same tuple. So each number of co-occurrences is stored once in L and once in T. The nodes have pointer to each other so we can find all tuples in T in logarithmic time using L.

This data-structure T allows us to retrieve a random element of maximal occurrence in logarithmic time. By remembering the tree with the largest index in T containing tuples, which can be handled in constant time with all update operations, returning a random element from this tree can be performed in logarithmic time. First we find which non-empty tree in L has the largest index. We can simply keep track of this number in constant time during each update operation, because we only need to handle updates that change the number by one. Then we randomly choose a node position in the tree. The trees in T are fully balanced, therefore once we know what positions in the tree are filled with nodes, then we can use this information to select the node at that particular position from the tree in logarithmic time. First, we randomly select a position in constant time, then we find the node in L on that position in logarithmic time. In our implementation of the k-ary trees we use binary trees ($k = 2$).

Theorem 6. *Using a k-ary tree and an array of k-ary trees of size p as underlying data-structure for $\mathbb{C}$ in GI, the operations F and the update operations to keep the data-structure $\mathbb{C}$ up to date take $O(n^2 \cdot p \cdot \log(n \cdot p))$ time with space requirements $O(n^2 \cdot p \cdot \log(n \cdot p))$.*

Proof. Building L initially comes down to simply inserting all $O(n^2 \cdot p)$ elements in a k-ary tree in $O(n^2 \cdot p \cdot \log(n \cdot p))$ time. Once L has been created, T is initiated with p empty k-ary trees in $O(p)$. We can populate all the trees in T with all elements in L in $O(n^2 \cdot p \cdot \log(n \cdot p))$ because all trees in T contain at most $O(n^2 \cdot p)$ elements. So creating the initial structure takes $O(n^2 \cdot p \cdot \log(n \cdot p))$ in total.

To update the number of occurrences from o to o' of a tuple, first look the current number of occurrences o up in L in $O(\log(n \cdot p))$ and update the number of occurrences in L in $O(\log(n \cdot p))$ time. To keep T up to date, we have to remove the tuple from one tree in T and move it to another. Removing or adding a tuple takes $O(\log(n \cdot p))$ time, because we can find it in L in $O(\log(n \cdot p))$ time, and use the pointer to find the node in T in constant time. This needs to be

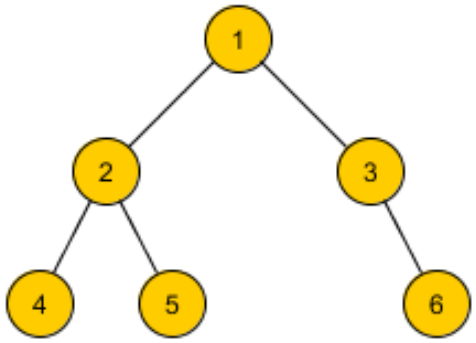

Figure 3.2: Example of a balanced tree that is not fully balanced.

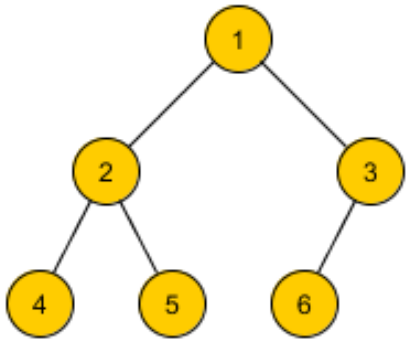

Figure 3.3: Example of a fully balanced tree.

performed for every update, so in total this takes $O(n^2 \cdot p \cdot \log(n \cdot p))$ time.

To find the maximal co-occurring set of tuples, we keep a pointer to tree T_H with the highest index in T. Every time we perform an update operation to the data-structure, we can keep this pointer up to date in constant time. The trees in L are fully balanced, therefore we know exactly where nodes are located in the tree and we can randomly select a node in any tree in T in $O(\log(n + p))$ time. Doing this for each rule takes in total $O(n \cdot p \cdot \log(n \cdot p))$ time.

Summing up the initial building of the structure, all the update operations and finding the most co-occurring tuples the total time complexity is $O(n^2 \cdot p \cdot \log(n \cdot p))$. All the trees together contain at most $O(n^2 \cdot p)$ elements, therefore the space requirements are $O(n^2 \cdot p \cdot \log(n \cdot p))$. □

In this section we have only derived theoretical bounds for GI. In Section 4.5 we will empirically show that the k-ary trees implementation improves the run-time scaling behaviour.

In conclusion, the array implementation will cost too much memory to be useful, and does not give a small worst-time complexity like the k-ary tree's implementation. The originally suggested hash table seems like a good idea. However, if the $\mathbb{F}$ operation of finding the maximal occurring tuple is taken into account then it turns out that it is large computational burden to perform this task. Moreover, the worst-case time complexity of amortized time is worse than the worst-case time complexity for the k-ary tree implementation. The implementation with the best worst-case time complexity we found was a k-ary tree combined with an array of size p filled with k-ary trees, this implementation has a worst-case time complexity and space requirements $O(n^2 \cdot p \cdot \log(n \cdot p))$. Yielding the lowest worst-case time complexity $O(n^2 \cdot p \cdot \log(n \cdot p) + n \cdot p^2)$ for GI among the implementations explored.

3.1.4 Schema Search

Schema Search is a unit exploiting search algorithm, that searches the BB space that is defined by the variational units. Similarly to OM in GOMEA, Schema Search also uses intermediary fitness function evaluations during mixing. Schema Search tries to overwrite the values of the original solution with the values of the variational unit. This continues until one of the variational units leads to an improvement, then the search is continued with the improved original solution until all variational units have been tested. The variational units are tested in random order in earlier versions or smallest first in more recent versions. Pseudo-code for this process is shown in Algorithm 3.

Each variant of SG uses Schema Search. More recent versions are not only looking for solutions that are strictly better, but also allow equal solutions. This can be very beneficial in problems with plateaus of equal fitness because it allows the algorithm to walk over fitness plateaus. If we consider using the

Algorithm 3 Schema Search. Pseudo-code was originally published by Cox et al. in [6].

Input: initial bit-string p_i
Input: Fitness function EvaluateFitness :: $\{0,1\}^n \rightarrow \mathbb{R}$
Input: variation symbol set Λ
Output: bit-string p_o

```
p ← p_i
for all λ ∈ RandomOrder(Λ) do
  candidate ← ExpandInto(p, λ)
  if EvaluateFitness(candidate) > EvaluateFitness(p) then
    p ← candidate
  end if
end for
```

FOS only for selecting BBs from the population and use these BBs to perform Schema Search instead of GOM, then we are performing a similar search like GOMEA. Now the BBs are are tested in reversed order and Schema Search has no Forced Improvement mechanism.

3.1.5 Variants of Schema Grammar

In this chapter we have introduced all the components required to build a SG search algorithm. We have discussed the model and a way to exploit this to do an evolutionary search. What remains for this section, is to show how these components can be transformed into various population-based variants of SG.

There have been 3 variants of SG proposed by Chris Cox; SG-Trap was originally proposed to solve Trap functions [5], SG-NK was proposed for NK-landscapes [6] and SG was proposed in the PhD Thesis of Chris Cox [7]. The latter is aimed at creating a BBO algorithm. In this work we will improve on this last variant with a new version of SG. This version will be discussed in the end of this section.

SG-Trap is aimed at solving various Trap functions [5]. SG-Trap rebuilds the population in each generation using the variational units of the previous iteration, in the first iteration TSs are used as variational units. To create an individual from the variational units, the variational units are randomly expanded over an initially empty solution until all variables are specified, then the Schema Search algorithm is executed to yield new offspring. When a new population is created, a new instance of the grammar is created and compressed with GI. The resulting schemata are used in the next iteration to build a new population and construct a new model with new variational units. The algorithm terminates if the global optimum was found, some computational budget has been reached or no more improvements are found.

The second variant of SG was proposed to solve NK-landscapes [6]. In contrast the SG-Trap, SG-NK uses a population of unique solutions. To create the

initial population, random solutions are generated and hill-climbed by executing Schema Search with only TSs. The resulting offspring are added to the initial population if they are unique. In each subsequent iteration the population is modelled as an instance of SG and compressed using GI. The resulting schemata are used as variational units for Schema Search, which is run on every individual in the population. The individuals in the resulting population are required to be unique, therefore duplicates are re-initiated by randomly expanding TSs to an empty solution until it is fully specified and running Schema Search with TSs as variational units. SG-NK terminates if the global optimum was found, some computational budget has been reached, or no more improvements are found.

In comparison to SG-Trap, SG-NK keeps a population of unique individuals. If the population allows duplicates then a few BBs that lead to very fit solutions these BBs can completely take over a population and the algorithm terminates prematurely. This can be attributed to the fact that all building blocks are tested in this variant, therefore if some of the blocks lead to very fit solutions, then all the good blocks will end up in most of the solutions. In addition, SG-NK does not rebuild the population in each iteration. So, if some genetic information is present before the model is built, but not contained within that model, then the information is lost in SG-Trap. In SG-NK this information may survive because the population is not rebuilt in each iteration.

In the most recent version of SG [7] each individual in the population is unique. Individuals are created from random bit-strings, which are then hill-climbed using the Greedy Ascent algorithm (shown in Algorithm 4). This algorithm is similar to a Best Improvement hill-climber. For Trap Functions SG uses a specialized Trap Function algorithm, shown in Algorithm 5. The most important difference between SG-Trap and SG-NK is that the search SG per-

Algorithm 4 Greedy Ascent.

Input: Problem size n
Input: Fitness function EvaluateFitness :: $\{0,1\}^n \to \mathbb{R}$
Output: Hill-climbed bit-string o

```
    o ← RandomBitstring()
    o.fitness ← EvaluateFitness(o')
    o' ← NULL
    while o' ≠ o do
 5:   o' ← o
      for all λ ∈ RandomOrder(n) do
        candidate ← FlipBit(λ, o')
        candidate.fitness ← EvaluateFitness(candidate)
        if candidate.fitness > o.fitness then
10:       o ← candidate
        end if
      end for
    end while
```

Algorithm 5 Specialized Hill-Climb Algorithm for Trap for SG.

Input: Problem size n
Input: Fitness function EvaluateFitness :: $\{0,1\}^n \rightarrow \mathbb{R}$
Output: Hill-climbed bit-string o
  $o \leftarrow$ RandomBitstring(n)
  $o.fitness \leftarrow$ EvaluateFitness(o)
  for all $\lambda \in$ RandomPermutation(n) **do**
    $candidate \leftarrow$ FlipBit(λ, o)
5: $candidate.fitness \leftarrow$ EvaluateFitness($candidate$)
    if $candidate, fitness > o.fitness$ **then**
      $o \leftarrow candidate$
    end if
  end for

forms is no longer looking for a strict improvement, but also accepts solutions with equal fitness. SG-NK is therefore able to walk on fitness plateaus in the fitness-landscape. The Schema Search algorithm gets a new parameter for the number of multi-scale hill-climbing steps m performed. Effectively this parameter describes the number of BBs to be tested. Like SG-NK, SG maintains a population of fit individuals. Instead of simply using the offspring pool and fresh individuals, SG increases selection pressure by performing truncation selection on the union of parents and offspring. SG, like SG-Trap and SG-NK, does not cache the fitness of any individual in the population. Whenever the fitness of an individual is required it is computed on the fly and as a result the fitness of solution O in line 15 is evaluated even though it has been previously. SG terminates if the global optimum was found, some computational budget has been reached or no more improvements are found. Complete pseudo-code for this Algorithm is shown in Algorithm 6.

In comparison with SG-NK, the truncation selection allows parents that did not survive Schema Search to be selected for the next generation. This allows SG to maintain diversity in the population better. Due to the limited number of BBs tested during Schema Search, a small number of good BBs among the BB pool does not necessarily lead to premature convergence in the offspring pool, because these good BBs are not selected every time for Schema Search.

We propose 3 modifications to SG and name the resulting version "SG v1.1" for clarification. Each time line 15 is executed, SG requires 2 fitness function evaluations to be performed. The evaluation of O is redundant because it has been previously evaluated. By caching the fitness of each individual in the population we require 50% less function evaluations in line 15. Secondly, the implementation with the hash table is not optimized for performing operation F (to compute the most frequent co-occurring tuple). By using a hash table, finding the most frequent occurring tuple requires all the tuples in the hash table to be iterated. Instead, we propose to use the k-ary trees implementation for $\mathbb{C}$,

Algorithm 6 Schema Grammar (SG). Pseudo-code was originally published by Cox et al. in [7].

Input: Population size p
Input: Fitness function EvaluateFitness :: $\{0,1\}^n \rightarrow \mathbb{R}$
Input: number of multi-scale hill-climbing steps m
Input: global optimum fitness f_{max}
Output: global optimum, or $\emptyset$ if not found

```
    ℙ ← ∅
    while |ℙ| < p do
      ℙ ← ℙ ∪ {HillClimb(EvaluateFitness, RandomSolution(n))}
    end while
 5: while max_{P∈ℙ} P.fitness < f_max do
      {V, G, S, Σ, R} ← GrammarInference(ℙ)
      Λ ← S ∪ Σ
      for all P ∈ ℙ do
        O ← P
10:     ℕ ← RandomSelection(Λ, m)
        //smallest-first
        for all λ ∈ SortByOrder(ℕ) do
          M ← ExpandInto(O, λ, R)
          if M ≠ O then
15:         if EvaluateFitness(M) ≥ EvaluateFitness(O) then
              O ← M
            end if
          end if
        end for
20:     ℂ ← ℂ ∪ {O}
      end for
      ℙ_NEW ← TruncationSelection(ℙ ∪ ℂ, p)
      if ℙ_NEW ∩ ℙ = ℙ then
        return ∅
25:   end if
      ℙ ← ℙ_NEW
    end while
    return argmax_{P∈ℙ} P.fitness
```

Algorithm 7 Schema Grammar v1.1 (SG v1.1).

Input: Population size p
Input: Fitness function EvaluateFitness :: $\{0,1\}^n \rightarrow \mathbb{R}$
Input: number of multi-scale hill-climbing steps m
Input: global optimum fitness f_{max}
Output: global optimum, or $\emptyset$ if not found

```
    ℙ ← ∅
    while |ℙ| < p do
      o ← RandomBitstring(n)
      o.fitness ← EvaluateFitness(o)
 5:   ℙ ← ℙ ∪ {o}
    end while
    while max_{P∈ℙ} P.fitness < f_max do
      {V, G, S, Σ, R} ← GrammarInference(ℙ)
      Λ ← S ∪ Σ
10:   for all P ∈ ℙ do
        O ← P
        ℕ ← RandomSelection(Λ, m)
        //smallest-first
        for all λ ∈ SortByOrder(ℕ) do
15:       M ← ExpandInto(O, λ, R)
          if M ≠ O then
            M.fitness ← EvaluateFitness(M)
            if M.fitness ≥ O.fitness then
              O ← M
20:         end if
          end if
        end for
        ℂ ← ℂ ∪ {O}
      end for
25:   ℙ_NEW ← TruncationSelection(ℙ ∪ ℂ, p)
      if ℙ_NEW ∩ ℙ = ℙ then
        return ∅
      end if
      ℙ ← ℙ_NEW
30: end while
    return argmax_{P∈ℙ} f(P)
```

described in Section 3.1.3. This implementation is optimized for the F operation which allows the worst-case time complexity for GI to be reduced from $O(n^3 \cdot p^3)$ or $O^*(n^3 \cdot p^2)$ to $O(n^2 \cdot p \cdot \log(n \cdot p) + n \cdot p^2)$. Thirdly, the Greedy Ascent algorithm is relatively expensive in terms of fitness function evaluations, because for a single bit-flip it requires $O(n)$ additional fitness function evaluations. These fitness function evaluations could be spent more efficient on multi-scale hill-climbing steps. For this reason SG v1.1 does not use a hill-climber to initiate the initial population, instead it starts with a population of random unique bit-strings.

3.1.6 Scaling of SG-Trap

In SG-Trap the Pre-terminal Symbols P are used as variational units, this is empirically shown to have a linear time complexity in terms of fitness function evaluations with a constant population size on Trap and hTrap [5]. Results of these experiments are shown in Figures 3.4 and 3.5, respectively. To determine the right population size, experiments were ran to determine the minimal population size to find all local optima. These results indicate that 80 is always enough for Trap functions, the results for the original experiment are shown in Figure 3.6. In Section 4.2.2 we will show that SG-Trap is able to reduce Trap functions to a Counting Ones problem.

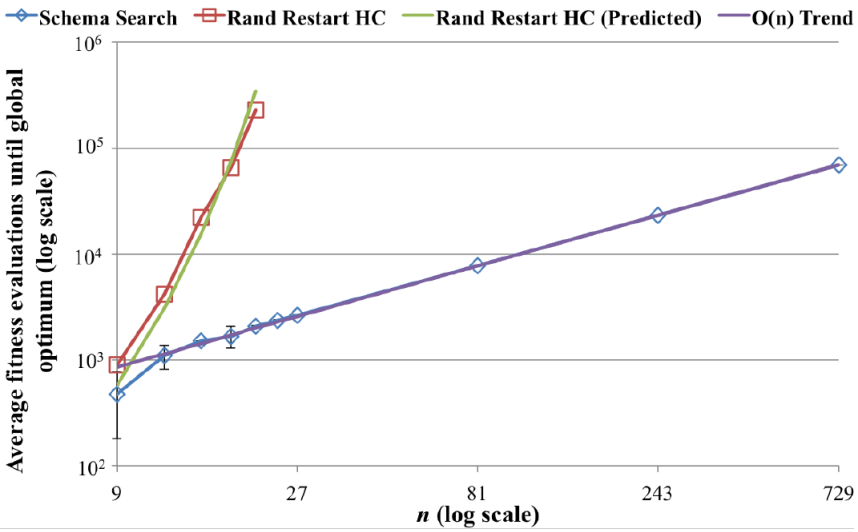

Figure 3.4: Average Fitness function evaluations on Trap functions with SG-Trap. The population size is set to 80. This image was originally published by Cox et al. in [5].

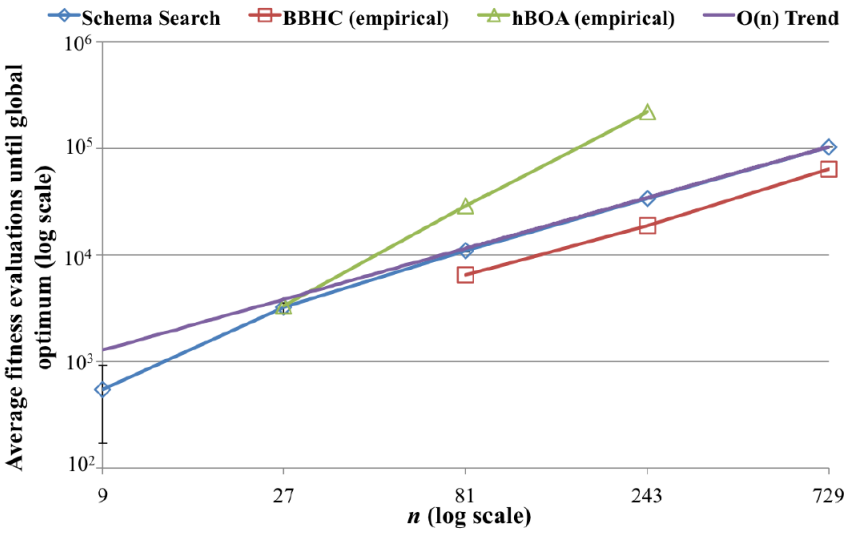

Figure 3.5: Average Fitness function evaluations hTrap functions with SG-Trap. The population size is set to 80. This image was originally published by Cox et al. in [5].

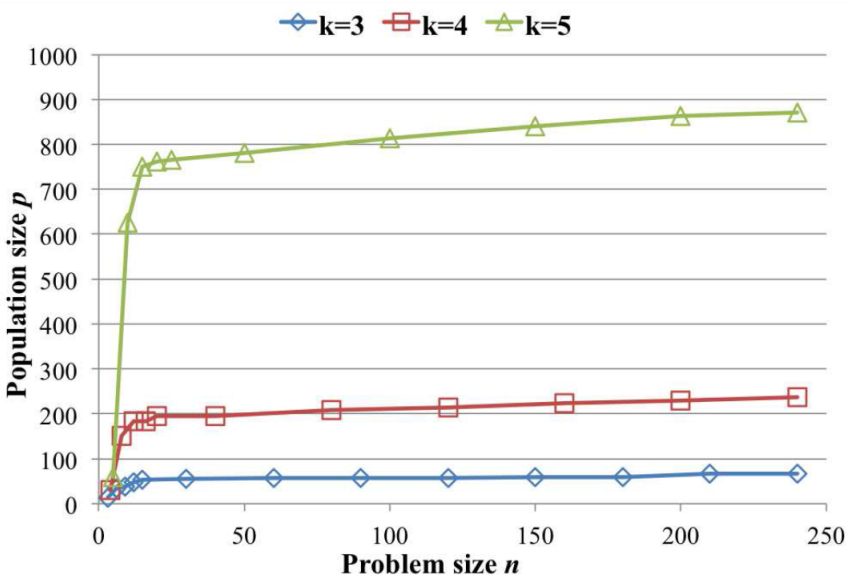

Figure 3.6: Minimal population size to correctly identify all local optima of a Trap function as pre-Terminal Symbols. This image was originally published by Cox et al. in [5].

3.2 Relation GOMEA

The algorithms SG and Gene-pool Optimal Mixing Evolutionary Algorithm (GOMEA) are both Black Box Optimization (BBO) algorithms that optimize problems in the discrete space. Both algorithms are population-based EAs that build a model in each iteration and use this model to search. The new solutions are used in the next iteration to build a model again. Both algorithms perform a neighbourhood search, they expand various BBs over solutions to create new solutions, guided by the model to find new solutions. However the approaches used in the model-building techniques differ greatly. LTGA hierarchically decomposes the problem as a Linkage Tree, built with the Variation of information (VI). All variants of GOMEA (including LTGA) model the problem as a Family Of Subsets (FOS). SG does not hierarchically decompose the problem in terms of a FOS structure, instead SG decomposes all solutions hierarchically in terms of BBs and compresses the population based on co-occurrence. The BBs used for this compression are used as variational units in the neighbourhood search. We explore the relation between these algorithms because these algorithms are both performing a neighbourhood search guided by a model, but the approaches used to construct these models and their use in an evolutionary search differ greatly.

The GOMEA framework can be considered an instance of MSS. MSS algorithms are formally defined using 4 features; Evolutionary Search, Redefining variational units, Separation of time-scales, and Diverse Sampling. GOMEA performs an evolutionary Search based on the current FOS called Gene-pool Optimal Mixing (GOM). GOMEA redefines variational units in each iteration by rebuilding the FOS and separates this process from GOM by doing them sequentially, similarly to SG. In addition, in GOMEA the population is initially composed of random bit-strings and during the algorithm diversity is maintained by using a selection method that allows weak individuals to survive (unlike truncation selection used in SG). Moreover, the tournament selection is performed on the union of parents and offspring in GOM-FI, thereby allowing parents that did not survive GOM-FI to survive nevertheless. With these 4 features GOMEA exhibits all features formally defined for MSS algorithms and the whole GOMEA framework is therefore a subset of MSS algorithms.

Due to the different approaches used in model-building, the representation of the problem is very different in both cases. Both algorithms hierarchically decompose the problem, but on different levels. SG hierarchically decomposes all solutions in terms of frequent occurring schemata, which is in sharp contrast to GOMEA, where the problem space is (hierarchically) decomposed in terms of a FOS structure. This different problem decomposition in SG leads directly to the ability to construct more complex models. For example, if variable A is dependent on variable B, but only if $A = 1$, then GOMEA would have no way to accurately decompose this relation, because it can not discriminate between dependencies that are only present when some variables have certain values.

In such a case, SG is able to model the correlations accurately in BBs that contain both variables A and B, by always setting A to 1. There is a down side to this enhanced problem representation, because it is more complex it is also computationally more expensive to build. For a population of size p, a problem of size n and cardinality c, the LT is constructed in $O(c^2 \cdot n^2 \cdot p)$ versus $O(n^2 \cdot p \cdot \log(n \cdot p) + n \cdot p^2)$ to create a compressed instance of SG.

In SG, the worst-case time complexity depends not on the cardinality of the problem, but solely on the amount of symbols to compress, given by the product of the population size p and the problem size n. The complexity of the model-building in LTGA depends on the cardinality c, because for computing the Variation of Information (VI), Equation 2.8, we need to consider all value tuples for each variable and count he number of occurrences to construct the Mutual Information (MI) matrix. The worst-case time complexity for model-building in SG does not depend on the cardinality of the problem. This is an indication that SG may have better run-time scaling behaviour with respect to cardinality. However, no research has been carried out to show that SG is able to solve higher cardinality problems. If, for example, the population sizes required is higher or the number of iterations required for solving higher cardinality problems with SG is a lot higher, then the run-time behaviour of SG is worse. The population sizes required for SG v1.1 are lower for problems with overlapping sub-functions than for LTGA, indicating that this is not the case.

When we compare performance of these algorithms, which is investigated in detail in Chapter 4, we see that the run-time for LTGA is significantly less, but run-time scaling behaviour is comparable although it is in favour of LTGA. The number of fitness function evaluations of SG on problems with disjoint sub-functions is lower, but the scaling behaviour of SG in terms of fitness function evaluations is comparable and in favour of LTGA on NK-landscapes. So in practice, the ability to express linkages in a more complex model is not necessarily beneficial in terms of fitness function evaluations, but increases run-time. In practice, it is therefore advised to first test LTGA, then only run SG if LTGA is not able to solve the problem (possibly because the problem requires a richer form of model representation expression) or if a fitness function evaluation is very expensive. In that case the amount of time won by having to compute less fitness can outweigh the increased run-time of GI. For higher cardinality problems the run-time for SG may become more competitive in terms of run-time to LTGA than it is for the binary problems tested in this work, because the model-building technique is not dependent on the cardinality of a problem, but only on the amount of symbols in the population (Section 3 contains proof on the computational complexity of GI).

In conclusion, GOMEA and all variants can be seen as instances of MSS. Both algorithms perform an unit-exploiting search and are similar except for the way the problem is hierarchically decomposed. In practice, LTGA is favoured and, unless computing a fitness function evaluation is expensive, it is the pre-

ferred method of choice. If computing a fitness function is expensive then it is unclear what algorithm performs better, because then the scaling behaviour of the number of fitness function is important, which only becomes clear after testing both algorithms on that particular problem.

3.3 Relation Frequent Pattern-Set Mining

Frequent Pattern Set Mining is a field of data mining, in which databases with an enormous number of transactions are mined for interesting patterns. Each of these transactions contains a subset of the set of all items. The task at hand is to find the set of interesting item-sets. Databases used are usually very large sparse databases with a lot of different items. Such an item-set or pattern is deemed interesting by a quality measure, various quality measures exist. Usually simply the number of transactions that contain each of the items in the frequent item-set is used as a measure and is referred to as the support of a pattern. Examples of such an algorithms are Apriori [27] and Krimp [8]. Apriori performs an exhaustive search to find all interesting frequent item-sets, whereas Krimp tries to re-encode the database to find create a loss-less compression by finding the set of frequent item-sets that compress the database best from a predetermined set of item-sets (possibly acquired using Apriori). Both these algorithms are looking for correlations or patterns in the data and this is precisely what GI does. GI looks for correlations to create a loss-less compression of the data. Moreover, these patterns can be considered building blocks, where we have a 1 if the item is present and 0 if the item is not present. In the remainder of this section these algorithms are considered as model-building alternatives to GI.

3.3.1 Apriori

Apriori is an algorithm originating from the field of Frequent Pattern-Set Mining [27]. It performs a exhaustive search to find every pattern or BB that is deemed interesting according to a quality measure. For example, if Apriori uses a minimal support of two as a quality measure, then all tuples in these BBs have a co-occurrence of at least two. Apriori starts with all singleton patterns for each variable and filters the ones that are not deemed interesting. In the subsequent generation Apriori will look for the patterns of size two, by combining every interesting singleton pattern of the previous generation and filtering the ones that are not interesting. In each subsequent generation the algorithm will look for the patterns of the size increased by one, combine sets that overlap exactly (size - 1) elements and finally filter all non-interesting patterns. This process repeats until no more interesting sets remain. In this way Apriori finds all interesting patterns in the database.

GI will produce $O(n \cdot p)$ BBs or patterns that are are able to hierarchically

model the population. Moreover, the result of GI is a schema hierarchy. Apriori returns a set of schemata that overlap and guarantees that this happens in a hierarchical fashion. For a schema F_i, $|F_i| < 1$ it is guaranteed that there is a pair of F_j, F_k, such that $F_j \cap F_k = \emptyset$ and $F_j \cup F_k = F_i$. However Apriori will have all subsets of F_i in the result. Therefore, Apriori is is able to hierarchically decompose the problem, but not in an efficient or useful way.

There's another drawback to this, there is a possibility that the number of frequent item-sets is $O(2^n)$. If this is the case, then computing all off them will be a large computational burden. GI only returns a linear number of item-sets $O(n \cdot p)$. If F_i is frequent then all subsets of F_i will be frequent as well. Testing all of these subsets would not only be a computational burden, but this would also result an needless testing of the fitness function.

A big drawback of Apriori compared to GI is that it scales poorly, GI scales with $O(n^2 \cdot p \cdot \log(n \cdot p) + n \cdot p^2)$ in theory, with a memory use $O(n^2 \cdot p \cdot \log(n \cdot p))$. For Aprori it is worse, Apriori can return an exponential number of schemata or patterns, which are linear in size in terms of the problem size. Simply reading the output requires $O(n \cdot 2^n)$ memory and run-time.

In conclusion, using only Apriori as a model-building technique results in an exponential number of schemata to test and therefore an increase of fitness function evaluations. It is clear that Apriori is not able to efficiently solve hierarchical problems, because it has no way to hierarchically decompose a problem and returns far more patterns than required. We can therefore expect it to perform worse than GI when used as a model-building technique, both in terms of model-building complexity as fitness function evaluations. Therefore, Apriori is not suited as a candidate to be tested as an alternative model-building technique for GI.

3.3.2 Krimp

Krimp uses a different way of selecting interesting patterns. Krimp does not use an measure that deems an item-set interesting. Instead, Krimp works with a large set of candidate pattern-sets and a database. The algorithm will keep track of a set of patterns $\mathbb{P}$ or BBs and considers adding the candidate pattern-sets one by one. If adding a candidate pattern to $\mathbb{P}$ leads to a better compression, indicated by the Minimum Description Length (MDL) measure, then it is added to $\mathbb{P}$ [8]. When all candidate patterns have been considered, the algorithm prunes each candidate set in $\mathbb{P}$ if removing it improves the compression. The initial set of candidate pattern-sets is acquired prior to running the algorithm. This set can be acquired using Apriori, by looking for all sets with a support ≥ 2.

Compared to Apriori, Krimp does not return as much BBs as Apriori. Siebes et al. report that on average only between 100-1000 frequent patterns are found after pruning using Krimp, but Krimp does require a large set of candidate fre-

quent patterns. So regardless of the relative small number of schemata resulting from the Krimp, the memory use is linear in terms of the number of candidate patterns and the number of candidate patterns scales exponential in terms of the problem size.

An important difference between Krimp and GI is that the problem decomposition in terms of schemata or frequent sets. In GI the problem is hierarchically decomposed and GI returns a schema hierarchy that decomposes the problem hierarchically. For Krimp there is no guarantee because the BBs are the result of an optimal compression.

In order to see if Krimp could solve NK-landscapes, we have tested Krimp as model-building alternative to GI. We ran similar bisection experiments as the experiments in Section 4.6. We use the a hill-climber to construct the population. The implementation used to perform this experiment was published by Siebes et al, and downloaded from the website of the Algorithmic Data Analysis group in Utrecht. To use Krimp to compress a population, we create a database with n columns and fill the rows with individuals. Each column describes a single variable and each row describes a single solution. This framework is able to solve small NK-landscapes ($n < 20$), the global optima were predominantly found by hill-climbing the initial population. But for median instances ($n \geq 25$) we were not able to find the optimal population size within one week. These results are an indication that the computational burden that results from looking for an optimal compression is so expensive that Krimp in its current form is not suited for solving BBO problems.

In conclusion, Krimp requires a set of candidate patterns that is potentially exponentially in size $O(2^n)$ this will cause run-time scaling problems similar to Apriori even before the algorithm is ran. Moreover, this optimized set of patterns that results from optimizing the compression using MDL is not guaranteed to be hierarchical. In fact, we will be spending an enormous amount of computational resources computing this optimal set of patterns, which can not hierarchically decompose the problem. Therefore, Krimp is not suited as a candidate to be tested as an alternative model-building technique for GI.

Chapter 4

Performance Analysis

In this chapter we test and analyse all published variants and versions of Schema Grammar (SG). SG-Trap in Section 4.2, SG-NK in Section 4.3 and SG in Section 4.4, and finally in Section 4.5 a new and improved version of SG is analysed and compared to LTGA, this version is named SG v1.1.

The results for Linkage Tree Genetic Algorithm (LTGA) are acquired using our own implementation of LTGA. This implementation builds the tree using the Reciprocal Nearest Neighbour Algorithm, starts with a population of random solutions and performs Gene-pool Optimal Mixing with Forced Improvements (GOM-FI). In Sections 4.4, 4.5 and 4.6 we will compare LTGA to versions of SG and to create as much consistency with SG, all versions of SG are parameterized with $m = 2n - 2$ to create as much consistency with LTGA as possible.

Implementations for all Schema Grammar experiments performed can be found at the SVN repository of the University of Utrecht.[1]

4.1 Methodology

All algorithms tested in this work have a population parameter p that describes the number of individuals in the population. We perform bisection experiments to find the minimal population sizes required for solving benchmark problems. A bisection is like a binary search without an upper-bound known beforehand. The bisection doubles the population size until it is able to solve a set of problem instances of the same problem size. After the bisection found an upper bound, it starts to refine the search by performing a binary search until the gap between the upper bound and lower bound is smaller or equal to some predetermined number, in this work this number is always one.

[1] Implementations for all Schema Grammar experiments found in this chapter can be found at the SVN repository of the University of Utrecht. The repository is named "edu.3253473.SchemaGrammar" and the code can be downloaded from "https://svn.science.uu.nl/repos/edu.3253473.SchemaGrammar".

To test whether a population size is large enough, we run a set of problem instances of the same size. We consider this set of problem instances solved if for a given population the optimum is reached at least 19 out of 20 times. Each of run is subjected to a limited computational budget, a single run of the algorithm is limited to 1,000,000 fitness function evaluations and a population size of 1024. This computational budget was chosen because LTGA is always able to solve the problems and problem sizes tested within these limits.

In order to get robust and reliable results for the bisection, bisections in this work are repeated 20 times and the results are averaged over to get a reliable estimate of the minimal population size to solve a set of problem instances of the same problem size. These averages are used to perform 100 runs of each problem size with the minimal population size mean to get an reliable estimate the number of fitness function evaluations required to solve a problem size.

Throughout this work all fitness function evaluations are counted (including all of the fitness function evaluations required to create the initial population). If for any reason we are averaging over runs that did not find the global optimum, then these runs are excluded. For example, if we use population sizes that directly resulting from a bisection then we can expect on average that $\frac{1}{20} = 5\%$ fails. When we report results we will exclude all runs that did not find the global optimum. So, we expect to average over the 95% successful runs, that is 95 runs.

In this work we fit the number of fitness functions with a polynomial $y = a + b \cdot x^c$. Unlike the function $g = a \cdot x^b$, the function y is not a straight line in log space, but will be straight line for large numbers of x (i.e. once $b \cdot x^c$ is significantly larger than a). In addition, the ranges of the axis in this work sometimes have very large ratio's, for example the y-axis may increase a 1000 fold while the x-axis only increases by a 10 fold. This in combination with the logarithmic scales used in this work causes very large differences in the plot for low numbers between the fitted line and the dot representing the mean.

All experiments in this work are all programmed in C# and ran on a 64-bit machine with 32 Cores, 2.6 GHz AMD Opteron 6282 SE, with 64 GB of memory.

4.2 SG-Trap

SG-Trap is introduced in [5]. Published work on SG-Trap only contains results for Trap and hTrap. The algorithm is discussed in detail Section 3.1.5. In order to analyse SG-Trap we first replicate the experiments published in Section 4.2.1 and we show that GI is able to reduce a Trap Function to a Counting Ones problem in Section 4.2.2. Finally, we test the algorithm in Section 4.2.3 on NK-landscapes and 2D Spin Glasses and compare this to LTGA.

4.2.1 SG-Trap: Trap Functions

The first experiment replicated is to see to what population sizes are required to correctly identify all local optima of the Trap functions. In this experiment the population is first initiated with random bit-strings, then Schema Search as defined in Algorithm 3 is executed for each individual with the TSs as variation operators. The resulting individuals are each composed of only local optima of Trap functions. To create a compressed instance of the grammar, the resulting population is compressed using the Grammar Inference (GI) algorithm, shown in Algorithm 2. A run is considered to be successful if each local optima of each Trap function is recognized as distinct symbol $s \in S$ in the grammar. To find the minimal population size required to accurately detect the local optima, the population size started at a low value and was then increased with a small step until 20 out of 20 runs correctly identified all the local optima as distinct symbols in the grammar. This experiment is performed for various values of n and k and repeated 10 times for each combination of n and k to get robust results, this is in contrast to the original experiment where it was performed once. The results are shown in Figure 4.1. The initial population size always started with 35, then increased this depending on the value of $k \in \{3, 4, 5\}$ with 5,10,20 respectively until the all 20 runs correctly identified all local optima.

Figure 4.1: Population sizes to accurately detect (20 out of 20 runs) the structure of Trap. The population size starts at 35 then is increased with steps of 5,10,20 depending on the value for $k \in \{3, 4, 5\}$ until the global optimum was found 20 out of 20 runs.

Compared to the original experiment, there is more variation in the replicated experiment. For this reason the experiment was required to be repeated 10 times to get robust results. The results reported in the original work show a very smooth line, as shown in Figure 3.6, without averaging over multiple runs. Despite the large amount of variation, the mean of the minimal required population sizes are comparable to the original work. Like in the original work, for the problem sizes tested the scaling of the minimal population size is sub-linear in n for any k and a population size of 80 is enough to accurately detect all the local optima among the problem sizes tested.

Why we see more variation here compared to the original results is unclear. Possibly the original publication was subjected to limited space requirements and the authors decided not to mention averaging over the results to save space, or only the points of a fitted line on the original results are shown. Another possibility is that the minimal population size was computed from smallest to largest population size, whilst starting with the population size found for the previous (smaller) problem size, and starting at an arbitrarily small population size for the smallest problem size. Performing the experiment in this way guarantees that the resulting line is smooth, because the population size can only grow or stay equal when the problem size is increased. Preliminary experiments have shown that performing the experiment in this way will increase the population sizes and they will be significantly larger than the original results. Despite the many possibilities what exactly is the reason for this large amount of variation compared to the original experiment is unclear.

In the second and third experiment from [5] SG-Trap was ran on Trap and hTrap of various problem sizes. The sub-linear scaling in the previous experiment allows the population size to be fixed at 80. Results for these experiments are shown in Figures 4.2 and 4.3. The results are comparable to what Cox et al. reported, which are shown in Figures 3.4 and 3.5. The results show that with a fixed population size an approximate linear scaling of fitness evaluations is achieved for both Trap and hTrap among the problem sizes tested. This is an indication that our implementation of SG-Trap is indeed correct.

Figures 4.4 and 4.5 also show fitness function evaluations versus the problem size, but to give insight in the variation the results are now shown as a boxplot. The variation at smaller problem sizes is higher, this can be attributed to the fact that the initial Schema Search with TSs on random bit strings already results in the global optimum. For larger problem sizes, the global optimum is found during the first Schema Search with variational units of the first model built, which contains all the local optima of each Trap functions as distinct schema symbols. In the next section we will show that SG-Trap is effectively able to reduce Trap to Counting Ones, if the set of Pre-terminal Symbols P are used as variational units and the population size is large enough.

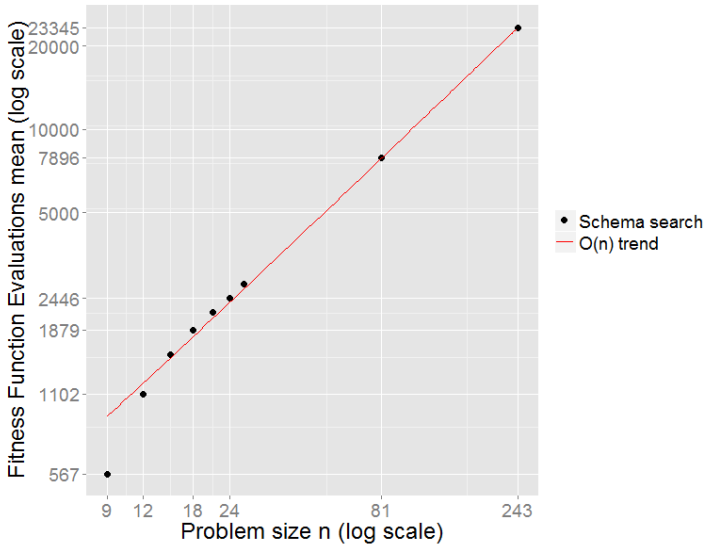

Figure 4.2: # Fitness function evaluation mean versus problem size on Trap functions using SG-Trap. The population size was set to 80.

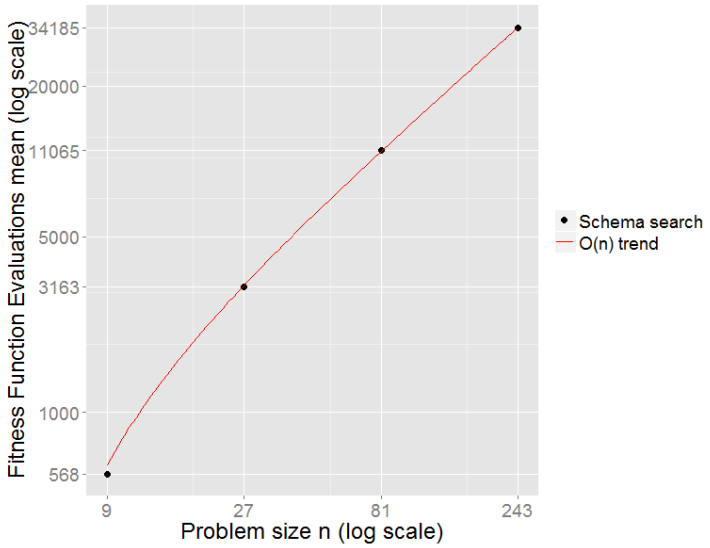

Figure 4.3: # Fitness function evaluation mean versus problem size on hTrap functions using SG-Trap. The population size was set to 80.

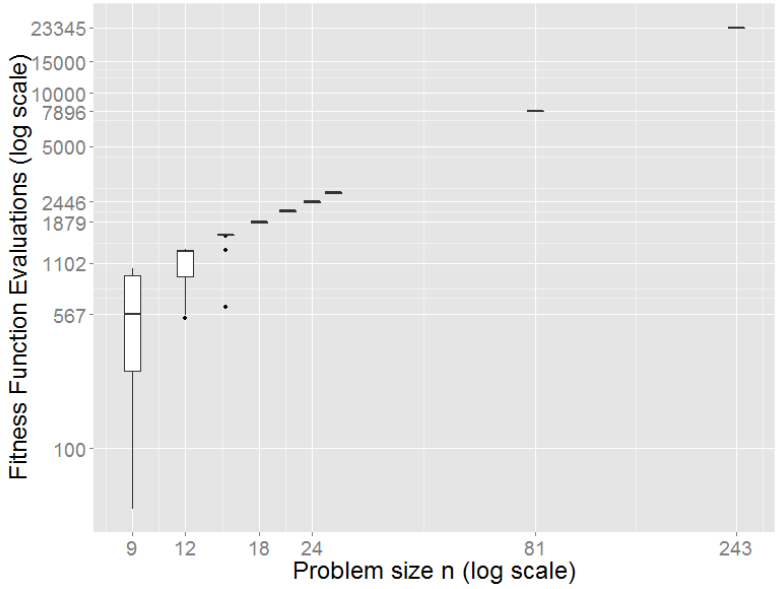

Figure 4.4: Fitness function evaluations versus problem size for Trap with $k = 3$ using SG-Trap. The population size was set to 80 in all runs.

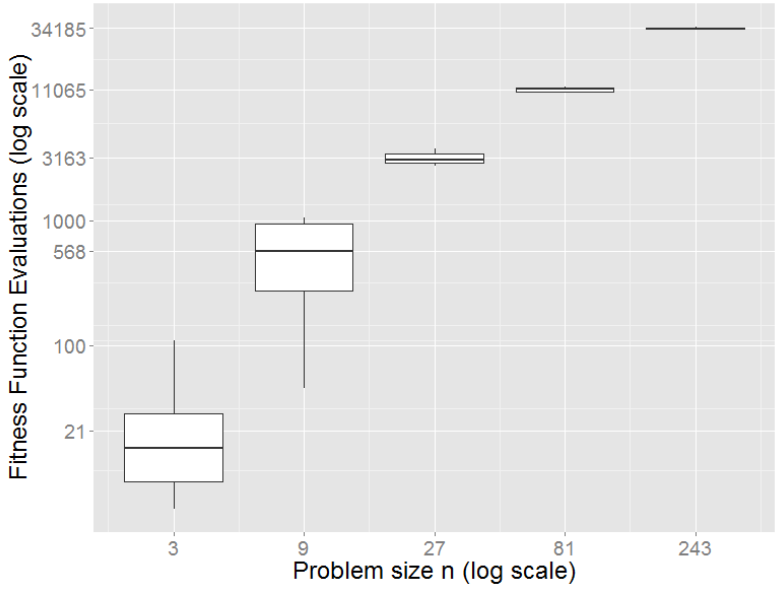

Figure 4.5: Fitness function evaluations versus problem size for hTrap with $k = 3$ using SG-Trap. The population size was set to 80 in all runs.

4.2.2 Counting Ones

The results of the first experiment indicate that a population size of 80 is enough for the problem sizes tested to always correctly identify all local optima as distinct symbols in the grammar. In this section we will show that if the population size is large enough then GI will always be able to identify all local optima as Pre-terminal symbols. GI will always solve the Trap in the first iteration of Schema Search when used in combination with the specialized Trap hill-climbing algorithm that finds a local optimum in linear time $O(n)$. We will show that SG solves a Trap with a linear number of fitness function evaluations.

If the population size is large enough, the population is compressed, and all local optima are recognized as TSs, then any smaller schema containing only a subset of values for a single local optima will be pruned away because only occur once in the grammar. From this we conclude that the set of Pre-terminal Symbols P is exactly the set of local optima for each Trap function involved. Let M is the total set of all Trap functions, and m_1 represents the global optimum of Trap function m composed of 1s and m_0 represents the local optimum of Trap function m composed of 0s, then P is given by:

$$P = \{p | m \in M, p = m_1 \vee p = m_0\} \tag{4.1}$$

Any larger schemata can only be composed of local optima because once all the local optima are found, the population is completely composed of schemata representing local optima and all TSs only occur once in their respective local optima. So, if we perform Schema Search with the Pre-terminal Symbols then we are simply doing a bit-flip hill-climb on a Counting Ones problem. For every Trap function we are only searching its 2 optima. Then we are effectively solving a Counting Ones problem because we only test two values for m functions.

In terms of fitness the problems will not be equal, but SG only takes the total ordering of solutions into account not their fitness. The relation between the fitness's is given by the following expression, where m is the number of Trap functions and o is the number of Trap functions with the global optimum: $f_{counting\ ones} = m \cdot f_{low} + (f_{high} - f_{low}) \cdot o = f_{Trap}$.

In SG-Trap we need a population of 80 to get a linear relation between number of fitness function evaluations and the problem size for a problem sizes tested. Initiating a population in SG-Trap takes $p + p \cdot 2n = 80 + 160n = O(n)$ fitness function evaluations. Doing the resulting bit-flips tests at most $2m = \frac{2n}{k} = O(n)$ local optima, yielding an overall linear number of evaluations. This only works as long as a population size of 80 is actually enough to correctly identify all local optima for the problem size tested.

In conclusion, if we only use the Pre-Terminal Symbols P as variational units in a Schema Search then we will effectively reduce the Trap function to a Counting Ones problem and find the global optimum in the first iteration of Schema Search. SG-Trap works extremely well on Trap functions. This may

be attributed to the fact that we are only searching with the smallest building blocks and this may work well on Trap functions or other functions with disjoint sub-functions and a small number of optima. The algorithm has no method to ensure diversity during evolutionary search, therefore the population may converge too quickly if SG-Trap is tested on problems with overlapping sub-functions. In the next section we will test this hypothesis and run the algorithm on 2D Spin Glasses and NK-landscapes.

4.2.3 SG-Trap: Overlapping Sub-functions

In this section the performance of SG-Trap on problems with overlapping sub-functions is tested. Bisections are performed with SG-Trap on 2D Spin Glasses and NK-landscapes.

The results for 2D Spin Glasses are shown in Figures 4.6 and 4.7. SG-Trap is able to solve 2D Spin Glasses but the population sizes required to solve 2D Spin Glasses are higher for SG-Trap than for LTGA. Both algorithms show a marginal increase or non-existing increase in population size when the problem grows from 64 to 81 bits. A possible explanation for this is that the set of 2D Spin Glasses with $n = 81$ is simpler to solve relatively than the sets of other problem sizes. When the algorithms are compared in terms of absolute fitness function evaluations, we see that for very small problem sizes SG-Trap is solving the problem more efficiently. SG-Trap requires less fitness function evaluations than LTGA for small problem sizes but scales worse. For small population sizes the hill-climb behaviour of SG-Trap dominates the execution of the algorithm, for this reason SG-Trap is able to beat LTGA on problems with overlapping sub-functions with small problem sizes. For 2D Spin Glasses of 49 bits and larger LTGA requires less fitness function evaluations and is therefore the preferred method of choice.

The results for NK-landscapes are shown in Figures 4.8 and 4.9. SG-Trap is able to solve small NK-landscapes, but with extremely large population sizes. With a limit on the population size the algorithm is only able to solve NK-landscapes of small sizes and for median sized problems the size of the population is required to be well over the pre-set limit of 1024. The large population size comes from the fact that SG-Trap has no way method to ensure diversity. Once SG-Trap performs an iteration with a neighbourhood that searches in the neighbourhood of one very fit solution, then all solutions will quickly converge to this one really good solution, and SG-Trap requires needlessly large population sizes to counteract this. These individuals are needless in the sense that LTGA can solve the problems tested with smaller population sizes. Therefore, this algorithm only full-fills two of the three keys to hierarchical success, it does not ensure diversity in the population. Like on 2D Spin Glasses, we see that SG-Trap is able to beat LTGA for small problem sizes on NK-landscapes, again this is in the region where hill-climbing dominates the execution of of SG-Trap.

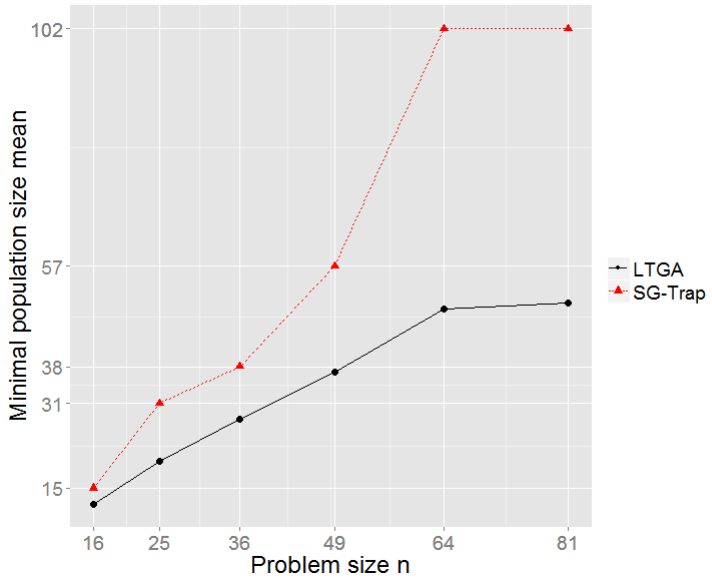

Figure 4.6: Population size versus problem size on 2D Spin Glasses using SG-Trap.

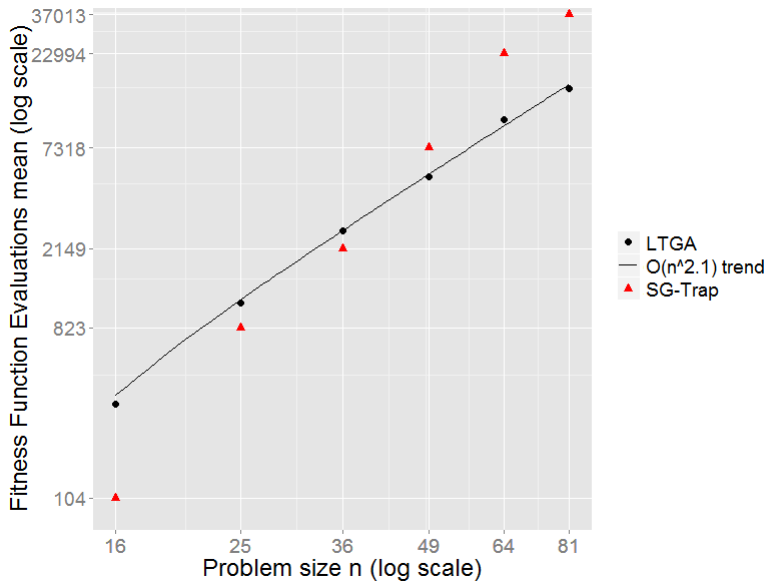

Figure 4.7: # Fitness function evaluation mean versus problem size on 2D Spin Glasses using SG-Trap.

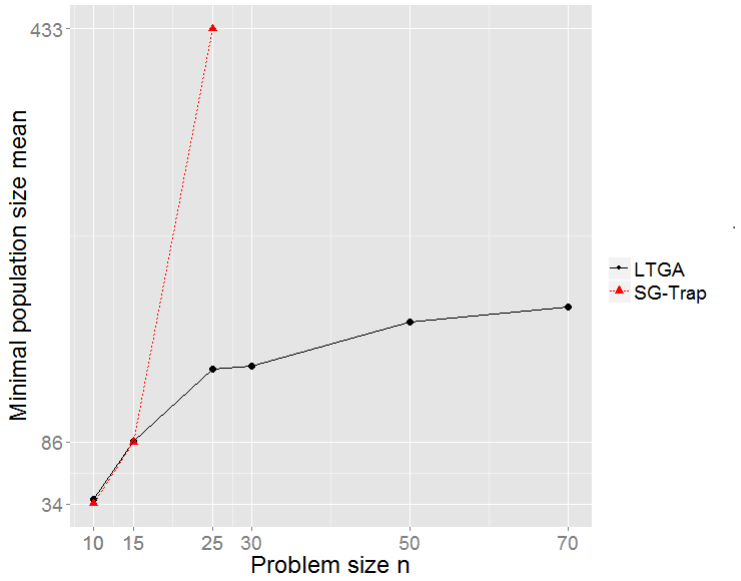

Figure 4.8: Population size versus problem size on NK-landscapes using SG-Trap.

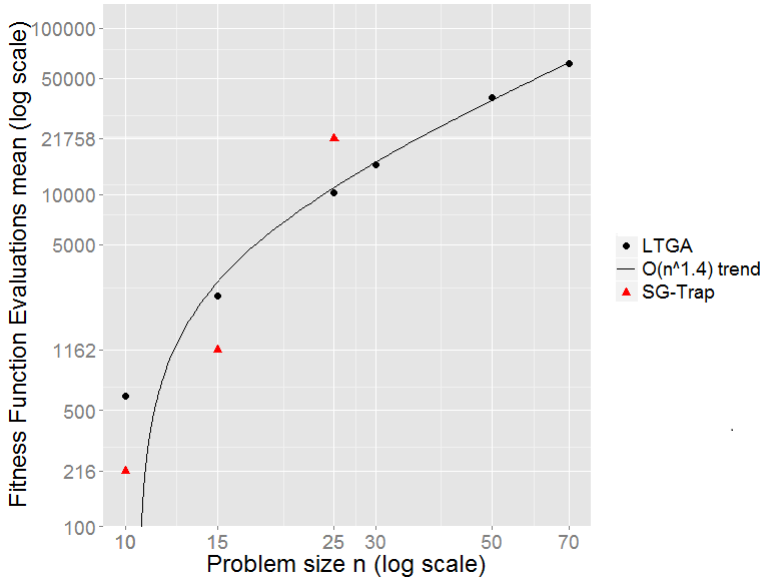

Figure 4.9: # Fitness function evaluation mean versus problem size on NK-landscapes using SG-Trap.

In conclusion SG-Trap is able to solve problems with disjoint sub-functions efficiently. On problems with overlapping sub-functions, SG-Trap is not longer able to ensure diversity and the performance breaks down. It is able to solve 2D Spin Glasses, but the population sizes required to solve NK-landscapes well exceed our pre-set limit of 1024. SG-Trap is not competitive with LTGA, because it only solves problems with non-overlapping sub-functions effectively.

4.3 SG-NK

In this section we analyse another variant of SG, SG-NK. This variant was originally proposed for solving NK-landscapes and it is claimed in [6] to be able to solve NK-landscapes efficiently. However, no actual results on NK-landscapes have been published. In order to gain insight on the algorithms performance, we test the algorithm on Trap functions, 2D Spin Glasses and NK-landscapes. Like in the previous section, bisection experiments are performed for each problem tested.

The first problem considered here are Trap functions, the results of the bisection are shown in Figures 4.10 and 4.11. SG-NK requires smaller population sizes than LTGA does. This can be attributed to the fact that SG-NK uses a population of unique individuals, whereas LTGA uses a population that is not required to be composed of unique individuals. The unique population in SG-NK does not allow for so many individuals used in population sizes for small problem sizes in LTGA. For example, if we consider hill-climbed Trap functions of size 9 then there are 3 sub-functions each with 2 local optima. In a population of unique individuals we have at most $2^3 = 8$ unique individuals, whereas LTGA can have $2^9 = 512$ different solutions and uses a population of 18 (non-unique) individuals. SG-NK hill-climbs is required to hill-climb more the reported 7 individuals because as new solutions are found, the chance to find unique solutions decreases. For example, if SG-NK has found 6 unique individuals on a problem size of 9 then the chance to get all 1s in a sub-function is $\frac{1}{4}$ and to get all 0s the chance is $\frac{3}{4}$. In the best case, if the algorithm did not find the solution composed of only 0s and one with only 1 sub-function with all 1s. The chance is then $1-\frac{3^3}{4^3}-\frac{1}{4}\cdot\frac{3^2}{4^2} = 0.4375$ to find a solution that is already in the population. This phenomena disappears when the problem size increases: for a problem size of 81 there would be 2^{27} possible outcomes, which means that there is at most $\frac{37}{2^{27}} = 2^{-21.8} \approx 0$ chance on finding an individual that is already in the population. For the problem sizes tested, LTGA is outperformed for the problem sizes tested by SG-NK, despite the better scaling behaviour with respect to the number of fitness function evaluations. Experimentation on larger problem sizes is required to show that this behaviour continues and LTGA is able to outperform SG-NK in terms of fitness function evaluations on Traps with larger problem sizes.

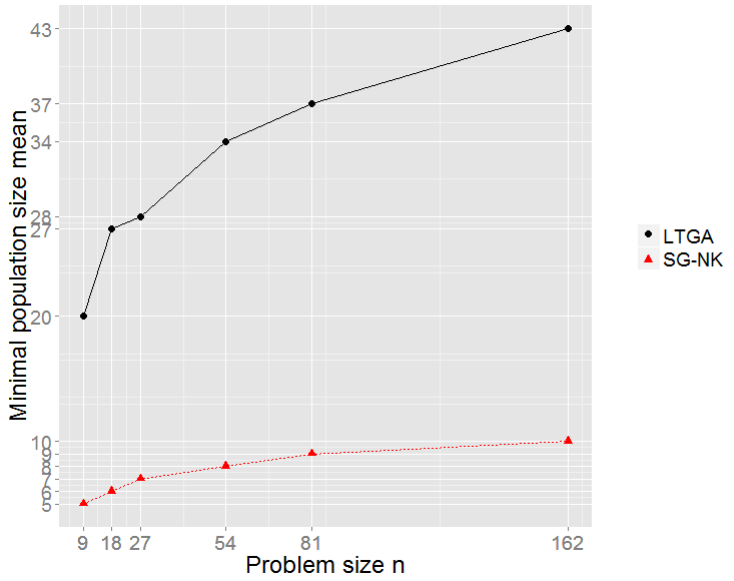

Figure 4.10: Population size versus problem size on Trap functions using SG-NK.

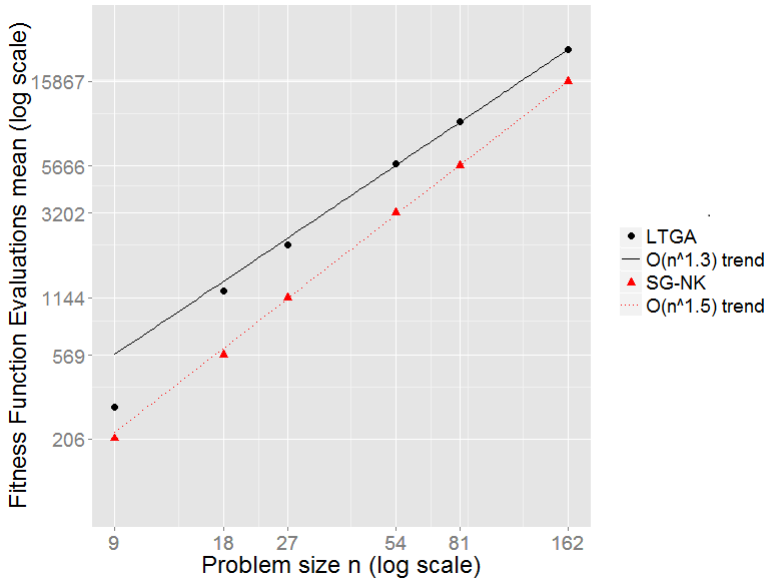

Figure 4.11: # Fitness function evaluation mean versus problem size on Trap functions using SG-NK.

We test SG-NK on 2 problems with overlapping sub-functions, 2D Spin Glasses and NK-landscapes. The results for these problems are similar. In Figure 4.12 the minimal population size mean versus the problem size for SG-Trap on 2D Spin Glasses is shown and in Figure 4.13 on NK-landscapes. In both figures we see that SG-NK uses very small population sizes, far smaller than both LTGA and SG-Trap. This can be explained by the fact that SG-NK uses a population of unique individuals and re-initiates duplicates. Performing evolutionary search in this way very small population sizes can already solve the problem, but this comes at a price in terms of fitness function evaluations. Each time the population is prematurely converging all duplicates are removed and replaced with new solutions containing new (evolutionary) information to be exploited. This is the reason that SG-NK is able to solve problems that SG-Trap can not, where SG-Trap prematurely converges, SG-NK enforces diversity by re-initiating all duplicates in the population.

In Figure 4.14 we see the number of fitness function evaluations mean versus the problem size on 2D Spin Glasses and in Figure 4.15 on NK-landscapes. Both figures show the better scaling behaviour of LTGA and the better performance on small problem sizes for SG-NK. This can be attributed to the fact that during the execution of SG-NK the hill-climb behaviour dominates the execution for small problem sizes. The hill-climb behaviour is more efficient than LTGA for small problem sizes, resulting in less fitness function evaluations than LTGA. The hill-climber is not able to solve large problem instances as efficiently as LTGA does. The scaling behaviour of LTGA is better than SG-NK and LTGA beats SG-NK on larger problem sizes.

In conclusion this algorithm is, in contrast to SG-Trap, able to solve all laboratory benchmark problems tested and with less fitness function evaluations than SG-Trap. SG-NK outperforms SG-Trap on all problems tested with overlapping sub-functions. SG-NK uses very small population sizes because duplicates are constantly replaced with random solutions. In comparison with LTGA, SG-NK has worse scaling behaviour. Practically better performing on Trap Functions, 2D Spin Glasses and small NK-landscapes for the problem sizes tested, but with respect to the scaling behaviour we can expect LTGA to outperform SG-NK on large problem instances. Therefore, in practice we can expect LTGA to solve the problem more efficiently, especially if the problem is composed of overlapping sub-functions.

4.4 Schema Grammar

In this section we analyse the most recent variant of SG proposed by Chris Cox [7] and compare our results to the results published. In this section we refer to the original results as ”SG (original)” or ”SG (ori)” and the results of our implementation of SG as ”SG (replicated)” or ”SG (rep). All bisection

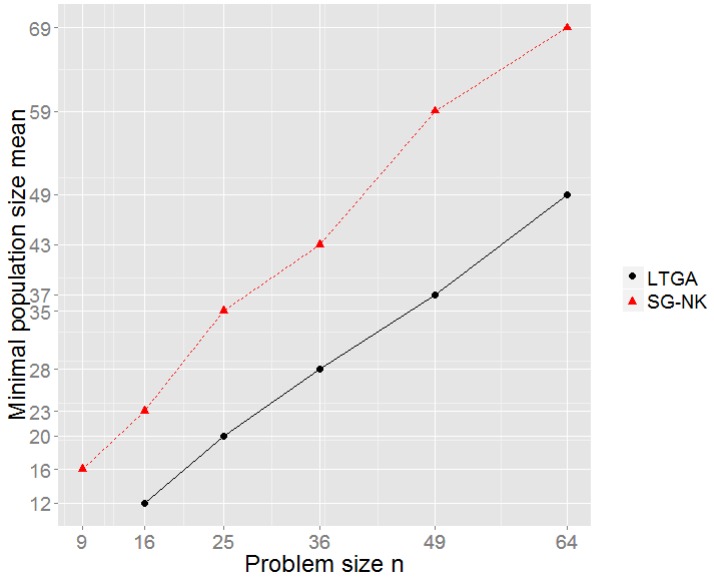

Figure 4.12: Population size versus problem size on 2D Spin Glasses using SG-NK.

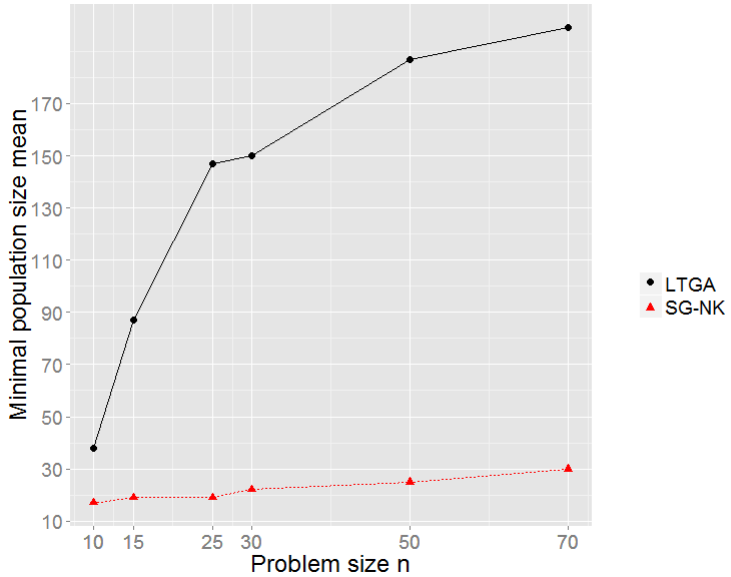

Figure 4.13: Population size versus problem size on NK-landscapes using SG-NK.

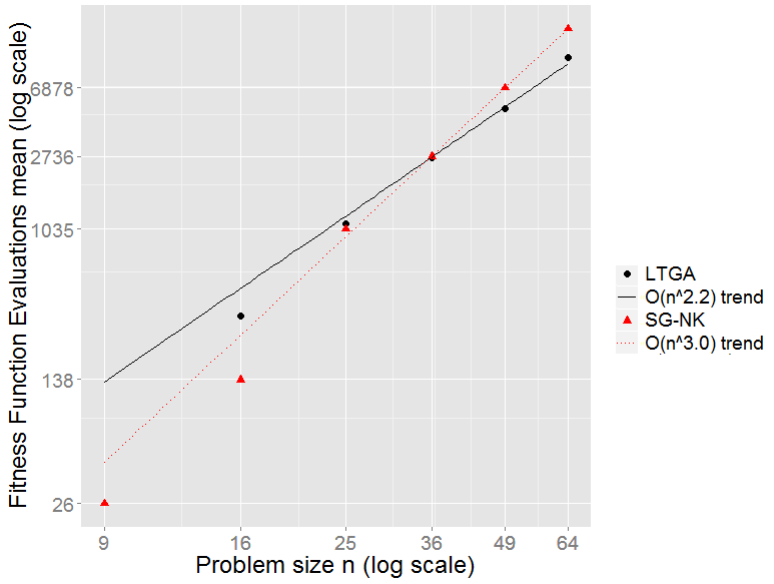

Figure 4.14: # Fitness function evaluation mean versus problem size on 2D Spin Glasses using SG-NK.

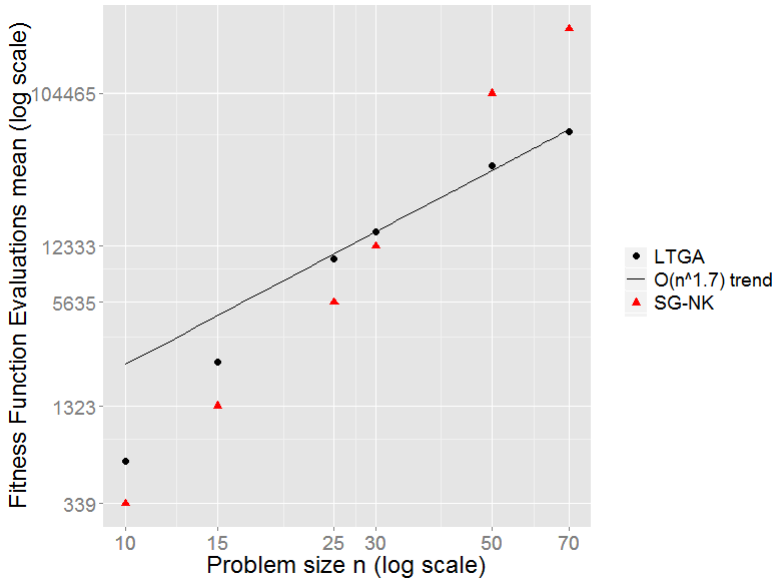

Figure 4.15: # Fitness function evaluation mean versus problem size on NK-landscapes using SG-NK.

experiments published in [7] are replicated in this section. The original research on SG contains too few details to clarify which hill-climbing method is used. We have tested several different hill-climbers (including First Improvement (FI) and Best Improvement) to replicate the original results but were not able too. E-mail correspondence cleared up some details, but even after clarifying which hill-climber was used for which test problem, we were not able to replicate the original results. The results that we report here are with the hill-climbing methods that were confirmed by Chris Cox. Our final implementation of SG uses a Best Improvement Hill-climber for 2D Spin Glasses and NK-landscapes, while using a First Improvement Hill-climber on Trap Functions that only tries to flip each bit once. The hill-climb algorithms are described in Section 3.1.5 as Algorithm 4 and Algorithm 5 respectively.

The results for the bisection on Trap functions are shown in Table 4.1, Figures 4.16 and 4.17. Here we see that our implementation of SG is able to solve Trap functions effectively. There is a sub-linear relation between the population size p and the problem size n. This is expected because there are no interactions between the sub-functions and we need only two occurrences of each local optima for GI to be able to detect the structure [5]. Our results for the population sizes we found are slightly smaller than what was reported in the original work, while the number of fitness function evaluations is almost a two-fold higher for some problem sizes. Even with the original population sizes, we were not able to replicate the original results. The large difference in fitness function evaluations suggests that our implementation is different from the original results, but we were able to replicate earlier results indicating our implementation is correct. We find similar scaling behaviour for our results compared to the original results. Another observation is that SG (replicated) uses relatively a lot of hill-climbing steps to find the optimum, especially for low problem sizes.

Table 4.1: Results from bisection using SG on Trap functions of various sizes. The measurements for SG (original) have been taken from the PhD thesis of Chris Cox [7].

	Population size		**Fitness function evaluations mean (s.d.)**		**Hill-climbing fitness function evaluations mean (s.d.)**
n	**SG (rep)**	**SG (ori)**	**SG (rep)**	**SG (ori)**	**SG (rep)**
9	7	7	425 (288)	295 (193)	397 (276)
18	16	17	1092 (395)	746 (188)	651 (123)
27	24	20	2398 (802)	1236 (324)	1048 (103)
54	33	27	7353 (1045)	3456 (452)	2417 (29)
81	36	30	12114 (1091)	5877 (669)	3927 (22)
162	42	37	28871 (2877)	14514 (1161)	9116 (31)
Estimated scalability			$O(n^{1.3})$	$O(n^{1.2})$	

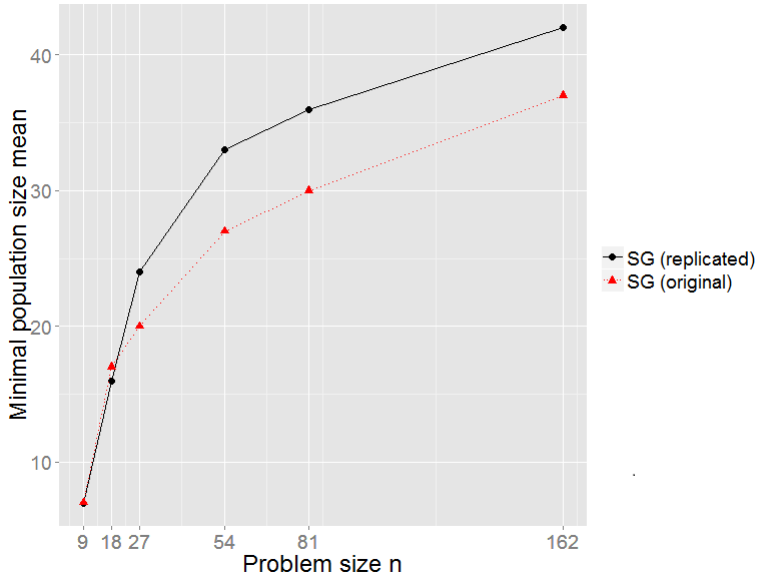

Figure 4.16: Population size versus problem size on Trap functions using SG.

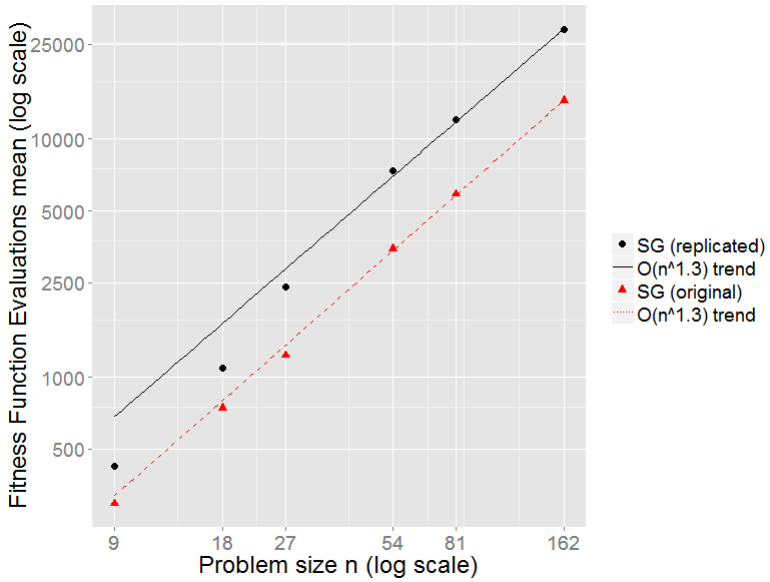

Figure 4.17: # Fitness function evaluation mean versus problem size on Trap functions using SG.

This is explained by the fact that SG regularly finds the global optimum for small problem sizes by hill-climbing. For larger problem sizes the variances decreases because the chance to generate an individual that is present in the initial population decreases.

The results for running SG on 2D Spin Glasses are shown in Table 4.2, Figures 4.18 and 4.19. SG is solving 2D Spin Glasses, but we were not able to replicate the original results. The population sizes are smaller, but the real distinction is in the number of fitness function evaluations into account. Our implementation uses significantly more fitness function evaluations than the original results. Moreover, simply counting the evaluations during the hill-climbing already exceeds the number of fitness function evaluations in the original results. From this we can conclude that at least the hill-climbing method used must be different from the original results. The original research did not contain a sufficient amount of details to replicate those particular hill-climbing algorithms. We were not able to replicate the original results although we used a hill-climber that was confirmed by Chris Cox and we have comparable results for SG-Trap on Trap and hTrap with this implementation of both GI and Schema Search.

The results for running SG on NK-landscapes are shown in Table 4.3, Figures 4.20 and 4.21. SG is able to effectively solve the problem, the population sizes we found for SG are significantly lower than in the original results, about half on average. However the performance in terms of fitness function evaluations is comparable. Interestingly, the number of fitness function evaluations of the algorithm are predominantly evaluations during the hill-climbing phase of the algorithm, not evaluations from Schema Search.

From the results we can conclude that SG is able to effectively solve the laboratory benchmark problems tested. In each problem tested either the population

Table 4.2: Results from bisection using SG on 2D Spin Glasses of various sizes. The measurements for SG (original) have been taken from the PhD thesis of Chris Cox [7].

	Population size		**Fitness function evaluations mean (s.d.)**		**Hill-climbing fitness function evaluations mean (s.d.)**
n	**SG (rep)**	**SG (ori)**	**SG (rep)**	**SG (ori)**	**SG (rep)**
16	6	8	247 (202)	140 (108)	211 (128)
25	11	11	1174 (862)	547 (376)	1006 (589)
36	15	20	4859 (2240)	2206 (1003)	3810 (1182)
49	22	27	14016 (3724)	5278 (1616)	11016 (1946)
64	29	34	30157 (5735)	10189 (2541)	24864 (3546)
81	25	40	44510 (4422)	17173 (3111)	35105 (1393)
100	34	45	89132 (7851)	25868 (3730)	71091 (2403)
Estimated scalability			$O(n^{2.7})$	$O(n^{2.0})$	

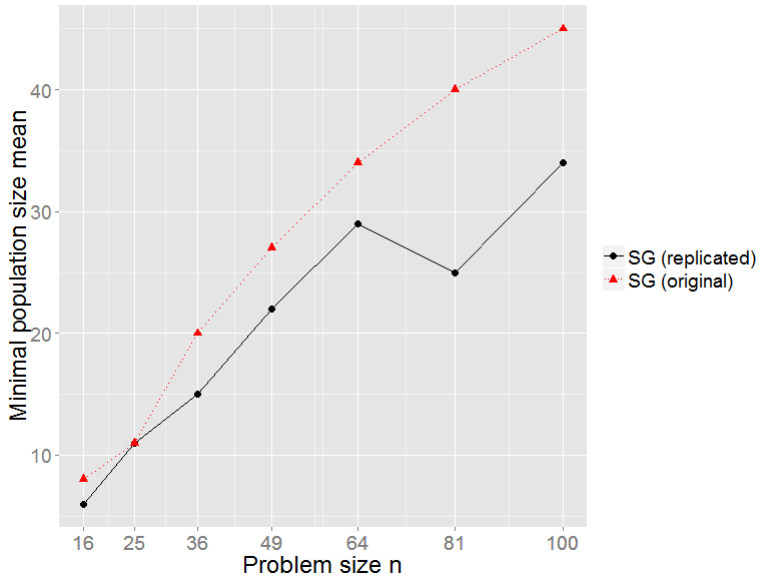

Figure 4.18: Population size versus problem size on 2D Spin Glasses using SG.

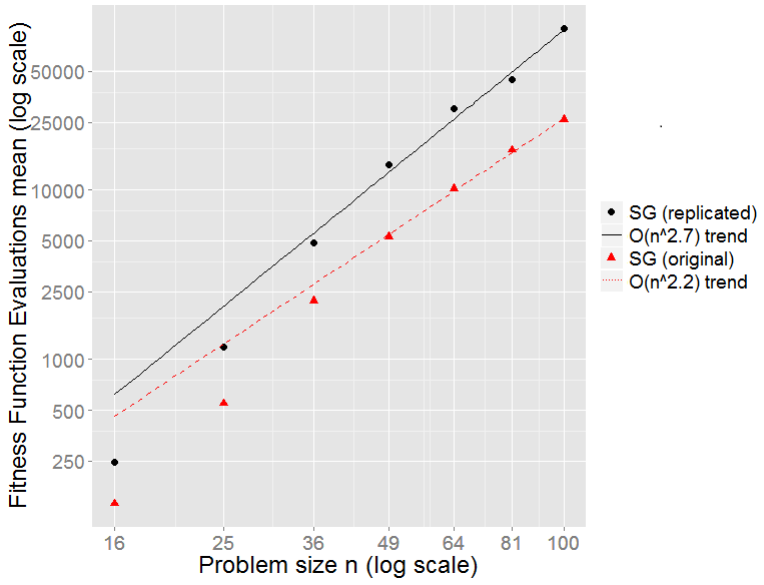

Figure 4.19: # Fitness function evaluation mean versus problem size on 2D Spin Glasses using SG.

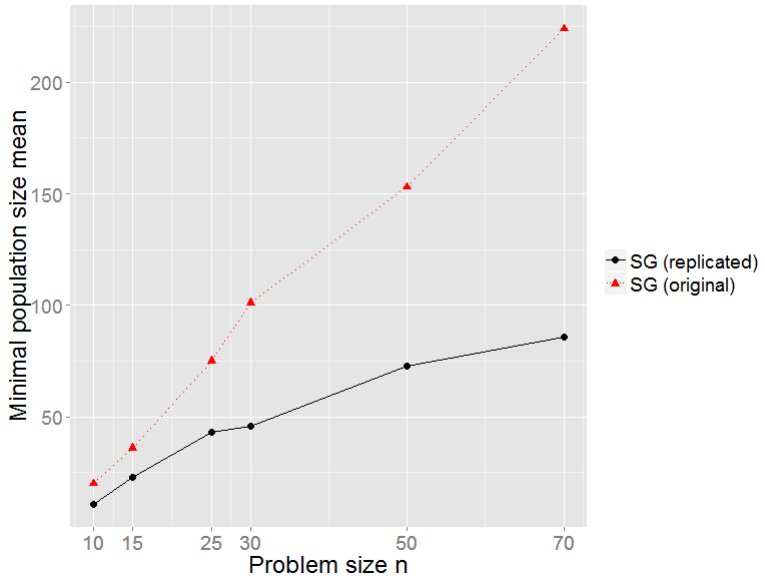

Figure 4.20: Population size versus problem size on NK-landscapes using SG.

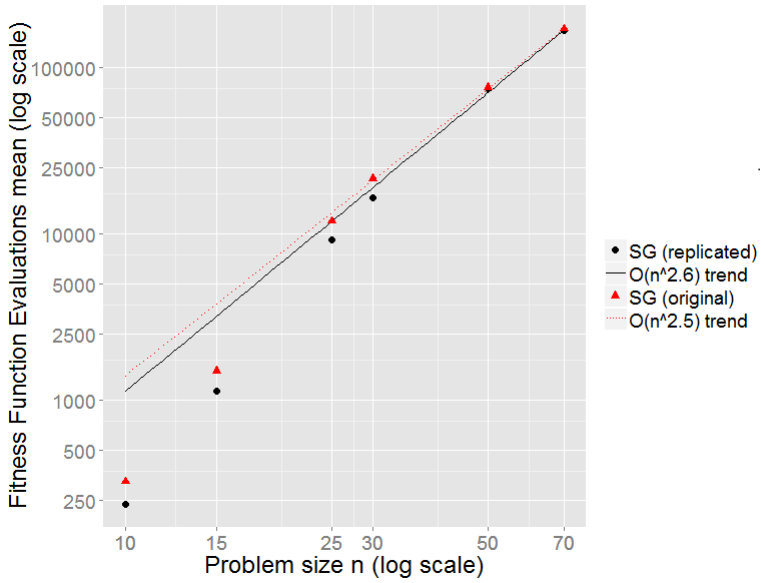

Figure 4.21: # Fitness function evaluation mean versus problem size on NK-landscapes using SG.

sizes tested or the number of fitness function evaluations were not comparable to previously published results. Unfortunately, it remains unclear what the difference is, in the case of 2D Spin Glasses we can be sure that the hill-climbing method used can not be the same because the large number of fitness function evaluations already exceeds original results. We were able to replicate earlier results that also use GI and Schema Search, so we can expect these implementations to be correct. Moreover, we were not able to replicate the original results because original research did not contain sufficient amount of details to be able to reproduce the them. Compared to SG-NK, SG is a significant improvement. It performs better in terms of fitness function evaluations for 2D Spin Glasses and NK-Landscapes. In contrast to SG-NK, SG does not rely on generating new individuals to add to the population after the initial population has been created.

There is room for improvement. For example, a large amount of computational resources is spent on hill-climbing random solutions to create the initial population but not on the Schema Search of SG. Additionally, a large chunk of computational resources is spent on computing the same solution twice. When we consider 2 iterations of the loop in Schema Search, then solution O is always evaluated twice, even though the second time the solution was unchanged. Each time the solution O is evaluated it has been previously evaluated in either the last iteration of SG or during hill-climbing when the initial population was created. These observations lead to the proposal of SG v1.1, which is analysed in detail in the next section.

Table 4.3: Results from bisection using SG on NK-landscapes of various sizes. The measurements for SG (original) have been taken from the PhD thesis of Chris Cox [7].

	Population size		**Fitness function evaluations mean (s.d.)**		**Hill-climbing fitness function evaluations mean (s.d.)**
n	**SG (rep)**	**SG (ori)**	**SG (rep)**	**SG (ori)**	**SG (rep)**
10	11	20	238 (193)	323 (303)	237 (192)
15	23	36	1131 (973)	1530 (1531)	1039 (808)
25	43	75	9300 (4630)	11970 (7482)	7076 (2826)
30	46	101	16520 (7845)	21661 (11381)	11290 (3341)
50	73	153	74780 (14762)	75992 (29840)	51686 (2231)
70	86	224	168196 (25307)	171490 (46441)	115705 (3452)
Estimated scalability			$O(n^{2.4})$	$O(n^{2.4})$	

4.5 Schema Grammar v1.1

In this section we test SG v1.1 on Traps, 2D Spin Glasses and NK-landscapes. The differences between SG and SG v1.1 are three-fold. First of all, SG uses a Best Improvement hill-climber, which is very expensive in terms of fitness function evaluations. These fitness function evaluations could better be spent on multi-scale hill-climbing steps. The fitness function evaluations are expensive because the hill-climbing algorithm requires $O(n)$ fitness function evaluations to find a single improvement (of potentially many improvements). Secondly, SG calculates the fitness of the same solution twice, namely each time solution O in line 15 of Algorithm 6 is evaluated even though it has been previously. Thirdly, we propose to use a different implementation for $\mathbb{C}$, we propose to use the k-ary trees implementation described in Section 3.1.3. This implementation improves the worst time complexity from $O(n^3 \cdot p^3)$ to $O(n^2 \cdot p \cdot \log(n \cdot p) + n \cdot p^2)$. Pseudo-code for this version is shown in Algorithm 7.

We have tested SG v1.1 on Trap functions, the results are shown in Table 4.4, Figures 4.22 and 4.23. The population sizes for SG v1.1, SG (replicated) and LTGA are comparable and each algorithm requires about the same amount of evolutionary information to solve the Trap. However, the population sizes for SG (replicated) are lower for small problem sizes. This is explained by the fact that SG (replicated) uses a hill-climber to initiate the population and the fact that number of local optima on Trap functions is smaller than the population size of SG v1.1 or LTGA.

The number of fitness function evaluations for SG v1.1 exceeds the number of evaluations that SG (replicated) requires. In addition, SG v1.1 is scaling worse. This can be directly attributed to the fact that SG v1.1 uses a specialized hill-climber for Trap functions that is very beneficial for solving Traps. Even without caching individuals during the multi-scale search, SG (replicated) is able to solve the problem with less fitness function evaluations than SG v1.1. LTGA scales

Table 4.4: Results from bisection using SG v1.1, SG (replicated) and LTGA on Trap functions of various sizes.

	Population size			**Fitness Function Evaluations mean (s.d.)**		
n	**SG (rep.)**	**SG v1.1**	**LTGA**	**SG (rep.)**	**SG v1.1**	**LTGA**
9	7	20	20	425 (288)	359 (228)	304 (122)
18	16	24	27	1092 (396)	1314 (338)	1237 (216)
27	24	28	28	2398 (802)	2392 (560)	2173 (312)
54	33	35	34	7353 (1045)	6643 (951)	5840 (241)
81	36	36	37	12114 (1091)	11440 (1437)	9759 (422)
162	42	43	43	28871 (2878)	30334 (1584)	23360 (807)
Estimated scalability				$O(n^{1.2})$	$O(n^{1.4})$	$O(n^{1.2})$

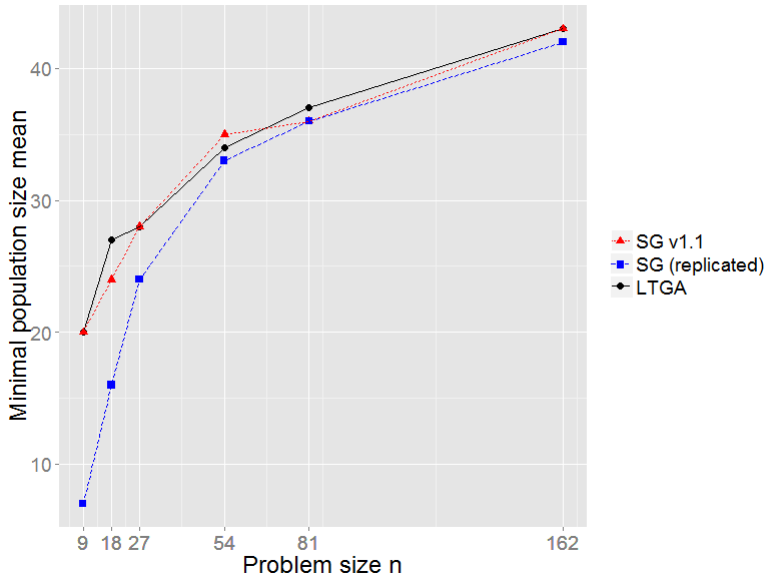

Figure 4.22: Population size versus problem size on Trap functions using SG v1.1, SG (replicated) and LTGA.

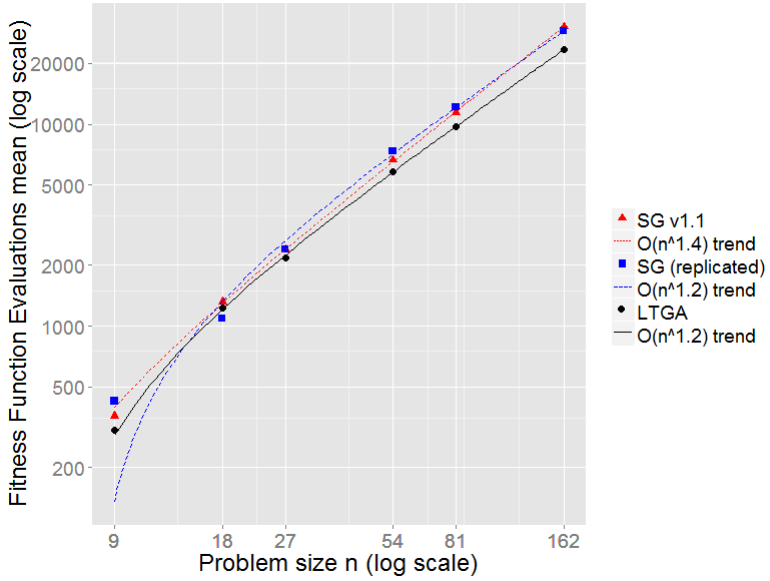

Figure 4.23: # Fitness function evaluation mean versus problem size on Trap functions using SG v1.1, SG (replicated) and LTGA.

better than SG v1.1 and with less fitness function evaluations for all problem sizes tested. SG (replicated) uses a specialized hill-climbing algorithm for Traps we can not consider this an BBO Algorithm, therefore the best BBO algorithm we find for Trap functions is LTGA.

We tested SG v1.1 on 2D Spin Glasses and NK-landscapes. The results for the minimal population size are shown in Figure 4.24 on 2D Spin Glasses and in Figure 4.25 on NK-landscapes. The minimal population size is significantly higher for LTGA than it is for SG v1.1, for 2D Spin Glasses there is a increase of about 50% and on NK-landscapes about 100%. The population sizes of SG (replicated) are higher than SG v1.1, this can be attributed to the fact that the Best Improvement Hill-climbing techniques reduces diversity too much before the evolutionary search is started, and to the fact that requires larger population sizes to avoid premature convergence.

In Figure 4.26 and Table 4.5 we see the number of fitness function evaluations on 2D Spin Glasses. The number of fitness function evaluations of SG v1.1 is significantly lower than of SG (replicated). The decrease in terms of fitness function evaluations is more than 50%, therefore this can not only be attributed to introducing caching. This is furthermore explained by the same reason why the population sizes of SG (replicated) are required to be higher, the local optima returned by the BI hill-climbing technique to initiate the population reduces diversity too fast for a efficient evolutionary search, this leads to larger population sizes and more fitness function evaluations. Compared to LTGA, SG v1.1 uses less fitness function evaluations than LTGA and scales marginally better, $O(n^{2.0})$ versus $O(n^{2.1})$.

In Figure 4.27 and Table 4.5 we see the results for the number of fitness function evaluations on NK-landscapes are shown. Like on 2D Spin Glasses, SG v1.1 improves over SG (replicated). SG v1.1 scales significantly better and with less fitness function evaluations for all problem sizes tested with $n \geq 25$. Like before, this is explained by the Best Improvement hill-climb technique that is used to create the population in SG. Unlike on 2D Spin Glasses, LTGA scales better than SG v1.1, but requires less fitness function evaluations for the problem sizes tested.

The ability to express richer linkages in the model does not necessarily lead to an improved performance. SG shows marginally better scaling behaver on 2D Spin Glasses, but LTGA scales significantly better on NK-landscapes. The performance of LTGA and SG v1.1 is similar in terms of fitness function evaluations. The reason why one algorithm performs better on one problem remains unclear. In both NK-landscapes and 2D Spin glasses, the neighbourhood of a single bit is 5 (including that particular bit). The 2D Spin glass has a lot of symmetry and ordered linkage over two dimensions, the NK-landscape do not have symmetry but has ordered linkage over one dimension. Possibly LTGA struggles with this multi-dimensional linking compared to SG, but it remains unclear why exactly one algorithm performs better on 2D Spin Glasses and the

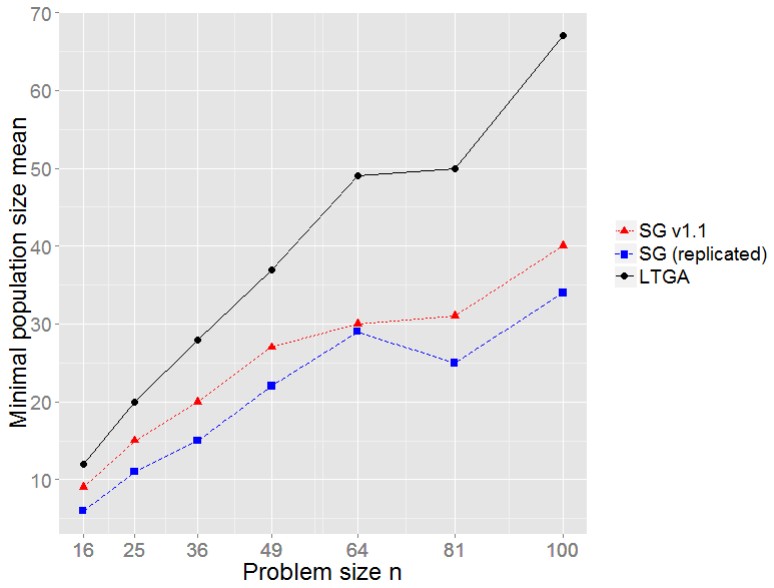

Figure 4.24: Population size versus problem size on 2D Spin Glasses using SG v1.1, SG (replicated) and LTGA.

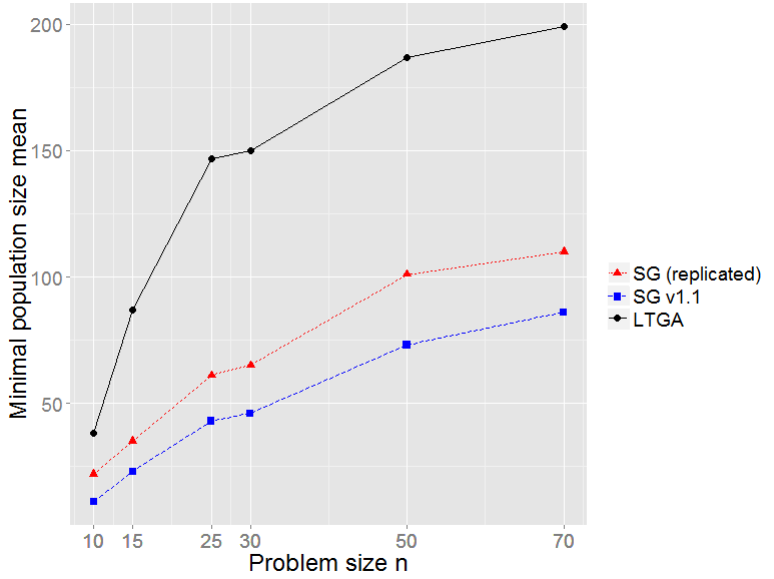

Figure 4.25: Population size versus problem size on NK-landscapes using SG v1.1, SG (replicated) and LTGA.

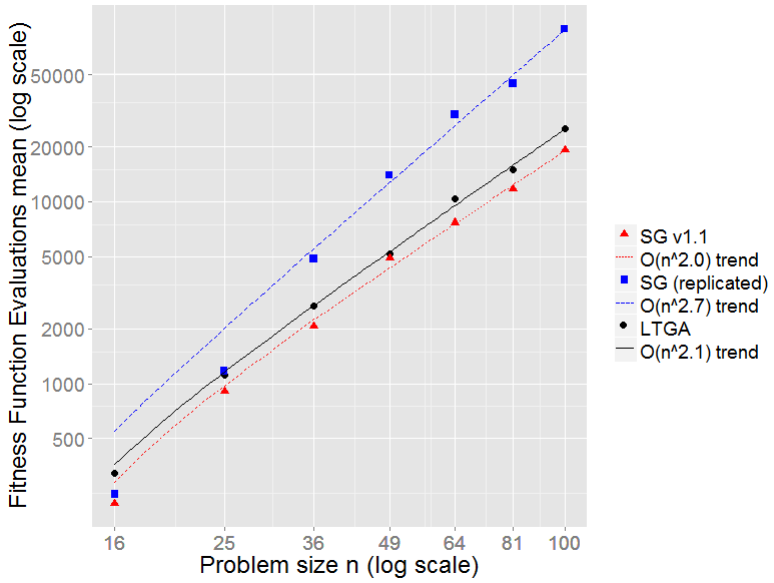

Figure 4.26: # Fitness function evaluation mean versus problem size on 2D Spin Glasses using SG v1.1, SG (replicated) and LTGA.

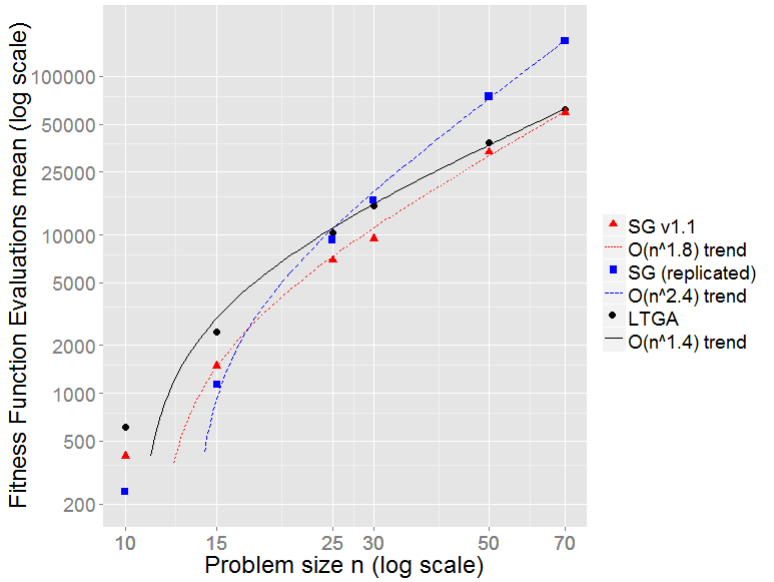

Figure 4.27: # Fitness function evaluation mean versus problem size on NK-landscapes using SG v1.1, SG (replicated) and LTGA.

other on NK-landscapes.

Table 4.5: Results from bisection using SG v1.1, SG (replicated) and LTGA on 2D Spin Glasses and NK-landscapes.

	Population size			**Fitness Function Evaluations mean (s.d.)**		
n	**SG (rep.)**	**SG v1.1**	**LTGA**	**SG (rep.)**	**SG v1.1**	**LTGA**
2D Spin Glasses						
16	6	9	12	247 (202)	218 (139)	323 (135)
25	11	15	20	1174 (862)	906 (431)	1113 (317)
36	15	20	28	4859 (2240)	2065 (695)	2685 (608)
49	22	27	37	14016 (3724)	4886 (1586)	5175 (1140)
64	29	30	49	30157 (5735)	7671 (2492)	10337 (2037)
81	25	31	50	44510 (4422)	11708 (3012)	15052 (2720)
100	34	40	67	89132 (7851)	19236 (4450)	25241 (3868)
Estimated scalability				$O(n^{2.7})$	$O(n^{2.0})$	$O(n^{2.1})$
NK-landscapes						
10	11	22	38	238 (193)	401 (329)	612 (329)
15	23	35	87	1131 (973)	1480 (843)	2448 (983)
25	43	61	147	9301 (4630)	6914 (3018)	10288 (2602)
30	46	65	150	16520 (78445)	9418 (2663)	15161 (2992)
50	73	101	187	74780 (14762)	33305 (10279)	38254 (5966)
70	86	110	199	168196 (25307)	59499 (17201)	61540 (7342)
Estimated scalability				$O(n^{2.4})$	$O(n^{1.8})$	$O(n^{1.4})$

In Figures 4.28 and 4.29 we see the run-time versus the problem size for 2D Spin Glasses and NK-landscapes. Running SG v1.1 with the k-ary trees implementation scales better than the hash table implementation and is indeed faster for the larger problem sizes tested. The run-time complexity on 2D Spin Glasses is improved from $O(n^{3.9})$ to $O(n^{3.0})$ and on NK-landscapes from $O(n^{3.2})$ to $O(n^{2.7})$. In both cases the complexity of the run-time is one polynomial degree larger than the number of fitness function evaluations, and we can not expect to do better in terms of run-time complexity (the number fitness function evaluations scales with $O(n^{2.0})$ or $O(n^{1.8})$ complexity, and a single evaluations takes linear time). From this we can conclude that there is a lot of overhead in the k-ary trees implementation for small problem sizes, but it shows to pay off to optimise the data-structure for operation $athbbF$ for larger problem sizes. Using this structure reduces the run-time scaling properties of GI, this can be attributed to the fact that this structure is optimised for performing the operation to find the most frequent tuple. The k-ary trees implementation scales better and requires less run-time for the larger problem sizes tested, therefore the k-ary trees implementation is the preferred implementation for $\mathbb{C}$.

On 2D Spin Glasses SG v1.1 performs better than LTGA, both in terms of fitness function evaluations and scaling behaviour. On NK-landscapes LTGA

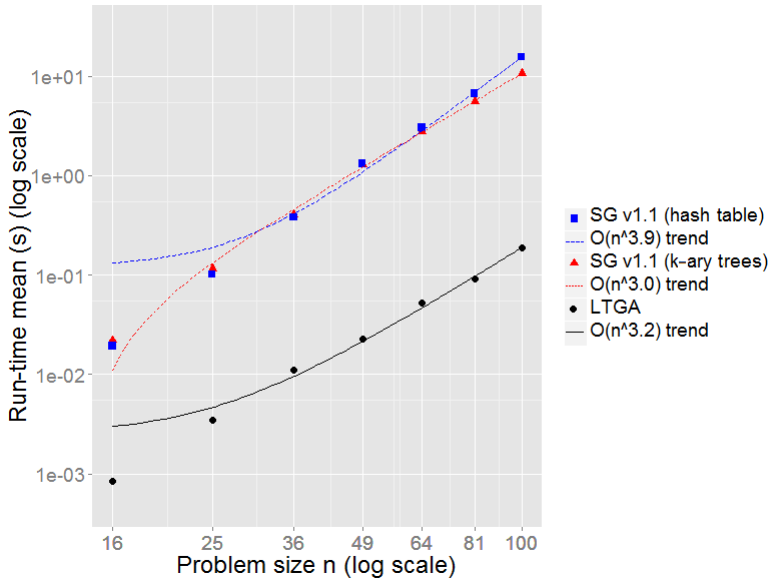

Figure 4.28: Run-time versus problem size on 2D Spin Glasses using SG v1.1 (k-ary trees), SG v1.1 (hash table) and LTGA.

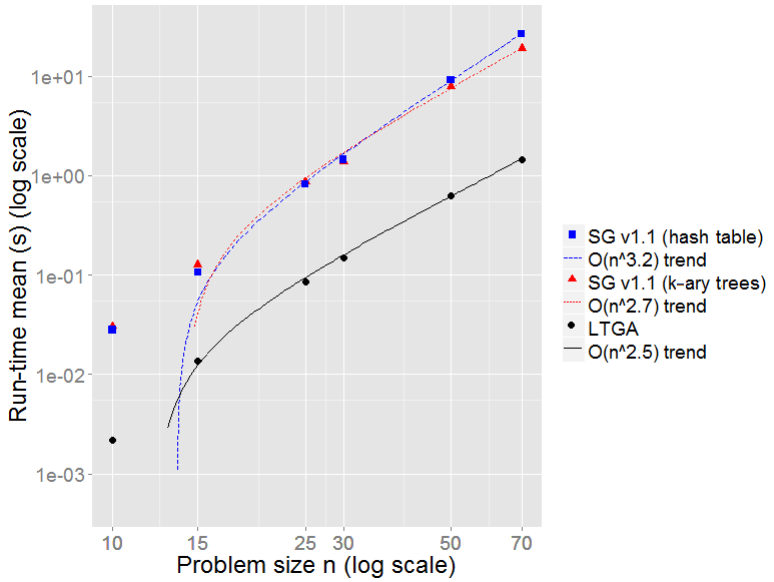

Figure 4.29: # Run-time versus problem size on NK-landscapes using SG v1.1 (k-ary trees), SG v1.1 (hash table) and LTGA.

performs better than SG v1.1, both in absolute fitness function evaluations as scaling behaviour. SG v1.1 always has a significantly larger run-time, there is a 10 to 50 times more in run-time for SG v1.1. It seems that the run-time scaling largely depends on how the number of fitness function evaluations scales. For both SG v1.1 and LTGA it is always about one polynomial degree of n more than the number of fitness function evaluations. This can be explained by the fact that a fitness function evaluation takes linear time to be computed. In fact, if one algorithm scales better than the other in terms of fitness function evaluations, then we can expect this algorithm to have a better run-time scaling behaviour despite the different computational complexities of the model-building techniques used. However, the improved run-time may only become apparent for SG on very large problem sizes because the run-time for the problem is 10 to 50 times more for SG v1.1 than it is for LTGA.

In practice, this means that whenever we need a BBO algorithm for the discrete space that the choice of the algorithm largely depends on the run-time of a fitness function evaluation. If it is not a computational burden, then we can expect LTGA to solve the problem faster. If evaluating a fitness function is a computational burden that dominates the computation, then choosing the algorithm which scales better in terms of fitness function evaluations will be beneficial. Unfortunately, the knowledge about which algorithm scales better on which problem is not known beforehand and both algorithms have to be tested on the problem to see which one scales better. If for any reason there is no time or willingness to test them both, then clearly LTGA is the preferred method of choice. The scaling behaviour of LTGA and SG is similar in terms of fitness function evaluations, although it may vary slightly from problem to problem. Running with LTGA is preferred due to the significantly lower run-time and the availability of parameter-less GOMEA [28].

4.6 SG v1.1 with FI

Originally, the search framework of SG was proposed with a Best-Improvement hill-climber. In this section we will test both algorithms, LTGA and SG v1.1, on populations that have been created using the First Improvement (FI) hill-climber, shown in Algorithm 8. This algorithm uses on average less fitness function evaluations to reach a local optimum. In addition, this algorithm may reduce the diversity of the population less because in comparison with Best Improvement hill-climbers, FI hill-climbers are less deterministic. Given a solution and a single iteration of Best Improvement will always select the next solution randomly among the set of best neighbours in the search neighbourhood, whereas FI picks the first neighbour encountered that is a improvement over the original solution. We will test the algorithms with FI to create the initial population on Trap, 2D Spin Glasses and NK-landscapes.

Algorithm 8 First Improvement Hill-Climbing Algorithm

Input: Problem size n
Input: Fitness function EvaluateFitness :: $\{0,1\}^n \rightarrow \mathbb{R}$
Output: Hill-climbed bit-string o

```
o ← RandomBitstring(n)
o.fitness ← EvaluateFitness(o)
o' ← NULL
while o ≠ o' do
  o' ← o
  for all λ ∈ RandomPermutation(n) do
    candidate ← FlipBit(λ, o')
    candidate.fitness ← EvaluateFitness(candidate)
    if candidate.fitness > o.fitness then
      o ← candidate
      BREAK {break out of for-loop}
    end if
  end for
end while
```

The results for testing the algorithms with FI on Trap are shown in Table 4.6, Figures 4.30 and 4.31. The population sizes for SG v1.1 are lower for small problem sizes but comparable for larger problem sizes. The fact that they are lower for smaller problem sizes can be attributed to the fact that the hill-climb algorithm finds the global optimum, or is able to construct very large portions of it. When we compare the algorithms in terms of fitness function evaluations, running the algorithms with a hill-climber on Trap requires more fitness function evaluations for all problem sizes tested, but with marginally improved scaling behaviour. For LTGA, hill-climbing decreases the population size significantly. This can be explained by the fact that using scaled Variation of Information measure gives very clear distinction between linked and not linked variables in the population composed of local optima, because the individuals are only composed of local optima of Trap functions. Both the number of fitness function evaluations and scaling behaviour are increased when using a hill-climber to create the population in LTGA. Therefore, when using LTGA on Trap, using a hill-climber is not beneficial. Like running the algorithms without a hill-climber, LTGA uses less fitness function evaluations to create the initial population. The scaling behaviour of both algorithms is comparable, and SG v1.1 uses larger populations than LTGA. The latter can be explained by the fact that after hill-climbing, inferring linkage between variables on problems with disjoint sub-functions is relatively easy, and by the fact that for SG v1.1 bijective mappings cause some local optima being ignored during mixing.

We have also tested both algorithms with FI to create the initial population on 2D Spin Glasses and NK-landscapes. The results for the population sizes are shown in Table 4.7 and Figures 4.32 and 4.33. For small problem sizes both

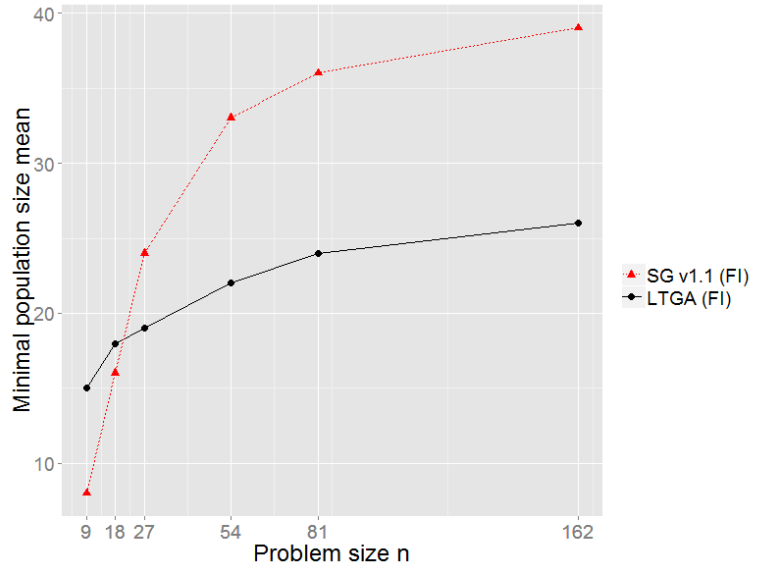

Figure 4.30: Population size versus problem size on Trap functions using SG v1.1 and LTGA with First Improvement hill-climbing to create the initial population.

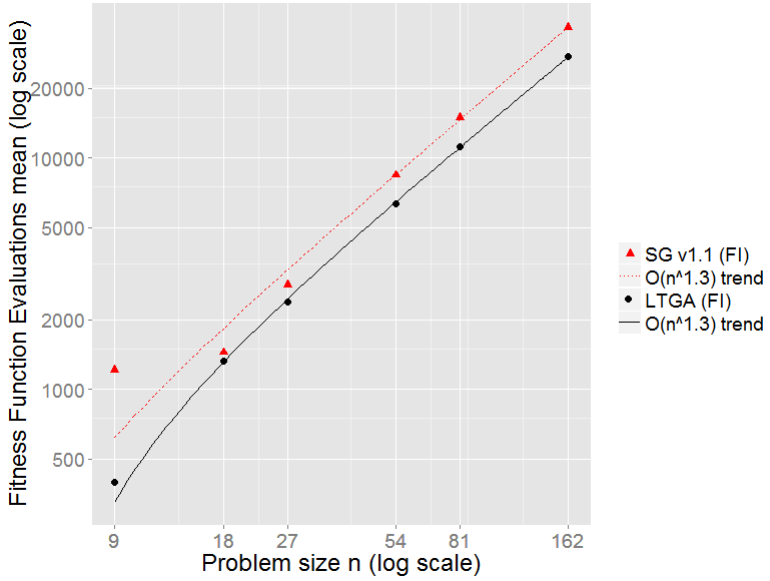

Figure 4.31: # Fitness function evaluation mean versus problem size on Trap functions using SG v1.1 and LTGA with First Improvement hill-climbing to create the initial population.

Table 4.6: Results from bisection using SG v1.1 with First Improvement hill-climbing to create the initial population on Trap functions of various sizes.

	Population size		**Fitness Function Evaluations mean (s.d.)**	
n	**SG v1.1 (FI)**	**LTGA (FI)**	**SG v1.1 (FI)**	**LTGA (FI)**
9	8	15	1211 (1226)	397 (100)
18	16	18	1446 (318)	1324 (140)
27	24	19	2931 (412)	2383 (159)
54	33	22	8445 (724)	6363 (218)
81	36	24	14990 (653)	11234 (425)
162	39	26	36807 (1771)	27364 (720)
Estimated scalability			$O(n^{1.3})$	$O(n^{1.3})$

SG v1.1 and LTGA require smaller population sizes than SG v1.1 and LTGA starting with a population of random individuals. This can be explained by the fact that the hill climber generally finds the global optimum or is at least able to construct a large portion of it. Like running the algorithms without a hill-climber, SG v1.1 uses significantly smaller population sizes than LTGA. This can be attributed to the fact that SG v1.1 uses a population of unique individuals.

The results for the number of fitness function evaluations are shown in Table 4.7 and Figures 4.34 and 4.35. Here we see that for both algorithms running them with a hill-climber to initiate the population increases the number of fitness function evaluations and scaling behaviour. For SG v1.1 run-time scaling is increased from $O(n^{1.8-2.0}$ to $O(n^{1.9-2.1})$, but for LTGA the increase is a more significant from $O(n^{1.4-2.1})$ to $O(n^{1.6-2.5})$. A large portion of the fitness function evaluations is spent on hill-climbing, and only checking after flipping a single bit. Another way of thinking about this is that the hill-climbers locate local optima too slow, effectively slowing down the evolutionary search in terms of fitness function evaluations.

We can conclude that running the algorithms with a hill-climber is not beneficial unless we are running on problems with disjoint sub-functions like Trap functions with SG v1.1. Possibly this is explained by the fact that on these types of problems linkage becomes very easy to exploit after Hill-Climbing. On problems with overlapping sub-functions using a First Improvement Hill-climber is not beneficial.

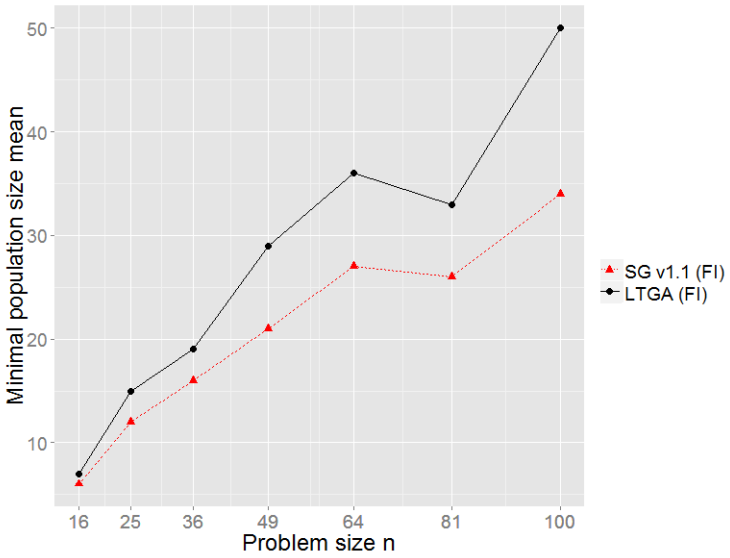

Figure 4.32: Population size versus problem size on 2D Spin Glasses using SG v1.1 and LTGA with First Improvement hill-climbing to create the initial population.

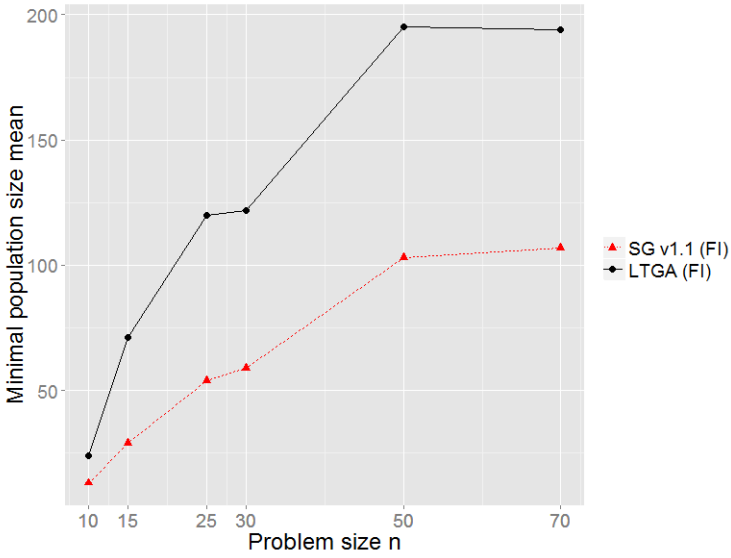

Figure 4.33: Population size versus problem size on NK-landscapes using SG v1.1 and LTGA with First Improvement hill-climbing to create the initial population.

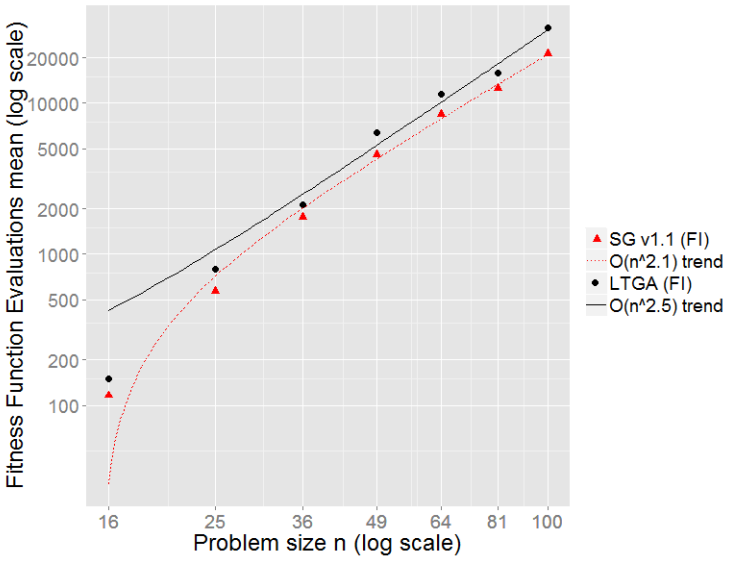

Figure 4.34: # Fitness function evaluation mean versus problem size on 2D Spin Glasses using SG v1.1 and LTGA with First Improvement hill-climbing to create the initial population.

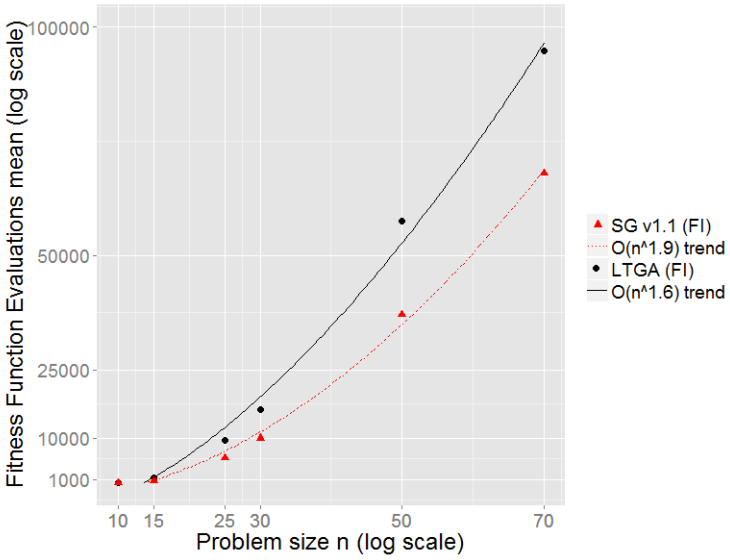

Figure 4.35: # Fitness function evaluation mean versus problem size on NK-landscapes using SG v1.1 and LTGA with First Improvement hill-climbing to create the initial population.

Table 4.7: Results from bisection using SG v1.1 and LTGA with First Improvement hill-climbing to create the initial population on 2D Spin Glasses and NK-landscapes of various sizes.

	Population size		**Fitness Function Evaluations mean (s.d.)**	
n	**SG v1.1 (FI)**	**LTGA (FI)**	**SG v1.1 (FI)**	**LTGA (FI)**
2D Spin Glasses				
16	6	7	116 (90)	149 (133)
25	12	15	567 (435)	799 (609)
36	16	19	1766 (886)	2126 (1092)
49	21	29	4564 (1201)	6362 (1678)
64	27	36	8413 (2323)	11438 (2598)
81	26	33	12478 (3392)	15823 (2549)
100	34	50	21197 (3363)	31320 (3736)
Estimated scalability			$O(n^{2.1})$	$O(n^{2.5})$
NK-Landscapes				
10	13	24	229 (227)	232 (203)
15	29	71	807 (688)	1280 (1334)
25	54	120	5687 (2879)	9486 (5250)
30	59	122	10014 (5459)	16207 (5217)
50	103	195	37026 (15941)	57634 (10332)
70	107	194	67996 (16441)	94771 (7254)
Estimated scalability			$O(n^{1.9})$	$O(n^{1.6})$

Chapter 5

Future Research

In this work we have shown that SG is able to effectively and efficiently solve laboratory benchmark problems and the complexity of the model-building has been reduced. Removing the needless parameters from the user perspective will be an important topic of future research. Examples of needless parameters from the user perspective are the population size p and the number of Multi-Scale hill-climbing steps m. If an end-user uses SG to solve a problem, they will not be interested in running experiments to find optimal values for parameters. Moreover, the user is only interested in solving the problem and preferably without any parameters that require optimization. Both LTGA and SG use different model-building techniques that result in a different but comparable evolutionary search. It is this different approach used that causes differences in performance. Finding out why one algorithm performs better on one problem will be an important topic of research.

5.1 New Problems

The only results in this work are limited to laboratory benchmark problems for GA. To truly test SG for real-world applicability the algorithm should be able to solve problems arising from real world problems. For example testing the algorithm on industrial benchmarks for the Maximal satisfiability (MAX-SAT) problem found for example in the SATLIB [29]. Another class of problems that is particularly interesting are higher cardinality problems because other BBO algorithms (including LTGA) have model-building techniques that scale quadratic in terms of the cardinality of the problem, for example LTGA $O(k^2 \cdot n^2 \cdot p)$, whereas GI scales with $O(n^2 \cdot p \cdot \log(n \cdot p) + n \cdot p^2)$. Moreover, GI is able to mode the problem on a value level it may be able to model and recognize structures that LTGA can not. For example if 2 variables are only correlated if one of them has a particular value, then in the best case LTGA

would have to chose between clustering them early in the tree or later. SG will be able to recognize the structure depending on a value for a specific variable because that is the level that SG models the problem. SG performs differently on 2D Spin Glasses and NK-landscapes than LTGA, this can be explained by the multi-dimensional linkage in 2D Spin Glasses. Perhaps running with 3D Spin Glasses will further separate the performance of the algorithms due to the multi-dimensional nature of this problem

5.2 Parameter-less Schema Grammar

In the world of BBO, the end-users that use BBO to solve problems are not interested in population sizes or the number of hill-climbing steps at all, they simply want to the algorithm to solve the problem preferably with as few parameters as possible. To solve these problems using SG one would have to learn the right population size by performing an expensive bisection experiment. It is important to use the right population size, because if the population size is to small the resulting solution may not be good enough, otherwise if the population size is to big we may be spending computational resources that might be better spent elsewhere. An alternative is to make an estimate of the right population size for the problem. It is very complex to come up with some sort of a equation to estimate the right population sizes and to do so for each problem is quite a burden. A more effective approach would be to remove the population size parameter completely and let a algorithmic technique handle population sizing. In recent publications from Peter Bosman, LTGA is tested with the population-size-free-scheme [30] and Willem den Besten proposed a variant of the algorithmic technique called P3 to remove the population parameter from LTGA [28]. In 1999 Georges Harik and Fernando Lobo propose an alternative approach in [31], they propose to have multiple populations of different sizes race to find the optimal solution which would remove the population size parameter. And if we count the number of fitness function evaluation to find the right population sizes as well then these population-free techniques prove to be a more efficient method. Before any recommendations can be suggested for removing the parameter for the number of hill-climbing steps, additional research is required to see how this parameter influences the algorithm exactly.

Chapter 6

Conclusion

Schema Grammar (SG) is an algorithm that uses Grammar Inference (GI) to build a model. GI compresses the population based on co-occurrence and is able to create multiple partially-overlapping schema hierarchies to model the problem. The resulting Building Blocks (BBs) from the model, correlations on a value level, are used to perform variation in an evolutionary search.

With this work we wish to answer the following question, are these Multi-Scale Search (MSS) algorithms, SG in particular, computationally efficient for solving BBO optimization problems with them? In order to answer these questions, we have analysed all known variants of SG on laboratory benchmark problems. SG-Trap is solving problems with disjoint sub-functions like Trap and hTrap in linear time, but can not ensure diversity causing performance to break down on problems with overlapping sub-functions like 2D Spin Glasses and NK-landscapes. SG-NK improves on SG-Trap by introducing a mechanism to ensure diversity. This variant is able to solve 2D Spin Glasses and NK-landscapes but not as efficient as Linkage Tree Genetic Algorithm (LTGA). The results for last variant proposed by Chris Cox SG were not reproducible because the research did not contain sufficient details. However, we were able to reproduce the general idea. In addition, we improved on the original results by proposing a new version named SG vl.1. This algorithm is able to solve all problems tested effectively and efficiently and improves SG on all problems tested and has comparable results to LTGA on 2D Spin Glasses and NK-landscapes. This shows that SG is able to effectively and efficiently solve BBO Optimization problems tested and is therefore an interesting candidate to be tested on problems originating from the industry alongside LTGA.

We have improved the model-building technique from $O(n^3 \cdot p^3)$ to $O(n^2 \cdot p \cdot \log(n \cdot p) + n \cdot p^2)$ by using k-ary trees as the underlying data-structure for GI.

We have explored the relation with LTGA. Both LTGA and SG perform a similar neighbourhood search, but the approach used to construct the vari-

ational units for the neighbourhood search is different. LTGA builds a Family of Subsets based on mutual information to model the problem, whereas SG uses a compression technique to compress the model and extract linkage from the correlations used in the compression. The GOMEA framework can be viewed as a subset of MSS. In practise, LTGA is favoured due to the much lower runtime. If a fitness function evaluation is very expensive, then it is unclear which algorithm will solve the problem faster because it is unclear when one algorithm performs better than the other on specific problems.

Additionally, we have explored the relation with pattern set mining and Grammar Inference (GI). These algorithms turn out to look for similar things. They are all looking for correlations in the data. These algorithms perform an exhaustive search to find all or some optimal set of patterns. This is not beneficial for an evolutionary search because this will lead to needless testing of the fitness function in addition to a needlessly large computational burden when building the model.

Acknowledgements

I would like to express my gratitude to my supervisor Dirk Thierens for introducing me with the subject, the useful comments, remarks and engagement through the learning process of this master thesis. I would like to thank my second supervisor Linda van der Gaag. Furthermore, I would like to thank Chris Sadowski for the useful discussions. Finally, I would like to thank my loved ones, who have supported me throughout entire process, both by keeping me harmonious and helping me putting pieces together. I will be grateful forever for your love.

Bibliography

[1] Alma As-Aad Mohammad Rahat, Richard M. Everson, and Jonathan E. Fieldsend. Multi-objective routing optimisation for battery-powered wireless sensor mesh networks. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 1175–1182. ACM, 2014.

[2] Dirk Thierens. The linkage tree genetic algorithm. In *Parallel Problem Solving from Nature, PPSN XI*, pages 264–273. Springer, 2010.

[3] Dirk Thierens and Peter A.N. Bosman. Optimal mixing evolutionary algorithms. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 617–624. ACM, 2011.

[4] Peter A.N. Bosman and Dirk Thierens. More concise and robust linkage learning by filtering and combining linkage hierarchies. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 359–366. ACM, 2013.

[5] Chris R. Cox and Richard A. Watson. Inferring and exploiting problem structure with schema grammar. *Parallel Problem Solving from Nature–PPSN XIII*, pages 404–413, 2014.

[6] Chris R. Cox and Richard A. Watson. Solving building block problems using generative grammar. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 341–348. ACM, 2014.

[7] Chris R. Cox. *Inferring and Exploiting Compact Models of Evolutionary Problem Structure*. PhD thesis, University of Southampton, 2015.

[8] Jilles Vreeken, Matthijs Van Leeuwen, and Arno Siebes. Krimp: mining itemsets that compress. *Data Mining and Knowledge Discovery*, 23(1):169–214, 2011.

[9] Kalyanmoy Deb and David E. Goldberg. Analyzing deception in trap functions. *Foundations of genetic algorithms*, 2:93–108, 1993.

[10] Richard A. Watson, Gregory S. Hornby, and Jordan B. Pollack. Modeling building-block interdependency. In *Parallel Problem Solving from Nature*, pages 97–106. Springer, 1998.

[11] Richard A. Watson and Jordan B. Pollack. Hierarchically consistent test problems for genetic algorithms. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 2. IEEE, 1999.

[12] Martin Pelikan and David E. Goldberg. Escaping hierarchical traps with competent genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 511–518, 2001.

[13] Stuart A. Kauffman. *The origins of order: Self-organization and selection in evolution*. Oxford university press, 1993.

[14] Martin Pelikan. Analysis of estimation of distribution algorithms and genetic algorithms on nk landscapes. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1033–1040. ACM, 2008.

[15] Lawrence Saul and Mehran Kardar. The 2d±j ising spin glass: exact partition functions in polynomial time. *Nuclear Physics B*, 432(3):641–667, 1994.

[16] Francisco Barahona. On the computational complexity of ising spin glass models. *Journal of Physics A: Mathematical and General*, 15(10):3241, 1982.

[17] Peter A.N. Bosman and Dirk Thierens. Linkage neighbors, optimal mixing and forced improvements in genetic algorithms. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 585–592. ACM, 2012.

[18] Martin Pelikan, Mark W. Hauschild, and Dirk Thierens. Pairwise and problem-specific distance metrics in the linkage tree genetic algorithm. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1005–1012. ACM, 2011.

[19] Peter A.N. Bosman and Dirk Thierens. The roles of local search, model building and optimal mixing in evolutionary algorithms from a bbo perspective. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 663–670. ACM, 2011.

[20] Martin Pelikan and David E. Goldberg. Hierarchical boa solves ising spin glasses and maxsat. In *Genetic and Evolutionary ComputationâĂŤGECCO 2003*, pages 1271–1282. Springer, 2003.

[21] Fabien Teytaud and Olivier Teytaud. Why one must use reweighting in estimation of distribution algorithms. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 453–460. ACM, 2009.

[22] Rob Mills, Thomas Jansen, and R. Watson. Transforming evolutionary search into higher-level evolutionary search by capturing problem structure. 2014.

[23] Richard A. Watson, Rob Mills, and Christopher L. Buckley. Transformations in the scale of behavior and the global optimization of constraints in adaptive networks. *Adaptive Behavior*, 2011.

[24] David E. Goldberg, Bradley Korb, and Kalyanmoy Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex systems*, 3(5):493–530, 1989.

[25] N. Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[26] John C. Kieffer and En-Hui Yang. Grammar-based codes: a new class of universal lossless source codes. *Information Theory, IEEE Transactions on*, 46(3):737–754, 2000.

[27] Gaurav Gupta Rupali. Apriori based algorithms and their comparisons. In *International Journal of Engineering Research and Technology*, volume 2. ESRSA Publications, 2013.

[28] W. den Besten. Parameter-less GOMEA. Master's thesis, University Of Utrecht, the Netherlands, 2015.

[29] Hoos H. Holger and T. Stützle. Satlib: An online resource for research on sat. In *Proceedings of Theory and Applications of Satisfiability Testing, 4th International Conference (SAT 2000)*, pages 283–292.

[30] Roy de Bokx, Dirk Thierens, and Peter A.N. Bosman. In search of optimal linkage trees. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 1375–1376, New York, NY, USA, 2015. ACM.

[31] Georges R. Harik and Fernando G. Lobo. A parameter-less genetic algorithm. In *GECCO*, volume 99, pages 258–267, 1999.