Chi-Square Matrix: An Approach for Building-Block Identification

Chatchawit Aporntewan and Prabhas Chongstitvatana

Chulalongkorn University, Bangkok 10330, Thailand
Chatchawit.A@student.chula.ac.th, Prabhas.C@chula.ac.th

Abstract. This paper presents a line of research in genetic algorithms (GAs), called building-block identification. The building blocks (BBs) are common structures inferred from a set of solutions. In simple GA, crossover operator plays an important role in mixing BBs. However, the crossover probably disrupts the BBs because the cut point is chosen at random. Therefore the BBs need to be identified explicitly so that the solutions are efficiently mixed. Let S be a set of binary solutions and the solution $s = b_1 \dots b_\ell,\ b_i \in \{0, 1\}$. We construct a symmetric matrix of which the element in row i and column j, denoted by m_{ij}, is the chi-square of variables b_i and b_j. The larger the m_{ij} is, the higher the dependency is between bit i and bit j. If m_{ij} is high, bit i and bit j should be passed together to prevent BB disruption. Our approach is validated for additively decomposable functions (ADFs) and hierarchically decomposable functions (HDFs). In terms of scalability, our approach shows a polynomial relationship between the number of function evaluations required to reach the optimum and the problem size. A comparison between the chi-square matrix and the hierarchical Bayesian optimization algorithm (hBOA) shows that the matrix computation is 10 times faster and uses 10 times less memory than constructing the Bayesian network.

1 Introduction

This paper presents a line of research in genetic algorithms (GAs), called building-block identification. The GAs is a probabilistic search and optimization algorithm [2]. The GAs begin with a random population – a set of solutions. A solution (or an individual) is represented by a fixed-length binary string. A solution is assigned a fitness value that indicates the quality of solution. The high-quality solutions are more likely to be selected to perform solution recombination. The crossover operator takes two solutions. Each solution is splited into two pieces. Then, the four pieces of solutions are exchanged to reproduce two solutions. The population size is made constant by discarding some low-quality solutions. An inductive bias of the GAs is that the solution quality can be improved by composing common structures of the high-quality solutions. Simple GAs implement the inductive bias by chopping solutions into pieces. Next, the pieces of solutions are mixed. In GAs literature, the common structures of the high-quality solutions are referred to as building blocks (BBs). The crossover operator mixes

M.J. Maher (Ed.): ASIAN 2004, LNCS 3321, pp. 63–77, 2004.
© Springer-Verlag Berlin Heidelberg 2004

and also disrupts the BBs because the cut point is chosen at random (see Figure 1). It is clear that the solution recombination should be done, while maintaining the BBs. As a result, the BBs need to be identified explicitly.

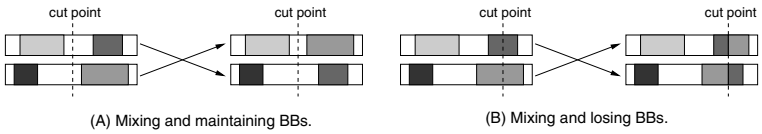


Fig. 1. The solutions are mixed by the crossover operator. The BBs are shadowed. The cut point, chosen at random, divides a solution into two pieces. Then, the pieces of solutions are exchanged. In case (A), the solutions are mixed while maintaining the BBs. In case (B), the BBs are disrupted

For some conditions [2, Chapter 7–11], the success of GAs can be explained by the schema theorem and the building-block hypothesis [2]. The schema theorem states that the number of solutions that match above average, short defining-length, and low-order schemata grows exponentially. The optimal solution is hypothesized to be composed of the above average schemata. However, in simple GAs only short defining-length and low-order schemata are permitted to the exponential growth. The other schemata are more disrupted due to the crossover. The trap function is an adversary function for studying BBs and linkage problems in GAs [3]. The general k-bit trap functions are defined as:

$$F_k(b_1 \dots b_k) = \begin{cases} f_{\text{high}} & ; \text{ if } u = k \\ f_{\text{low}} - u\frac{f_{\text{low}}}{k-1} & ; \text{ otherwise,} \end{cases} \tag{1}$$

where $b_i \in \{0,1\}$, $u = \sum_{i=1}^{k} b_i$, and $f_{\text{high}} > f_{\text{low}}$. Usually, f_{high} is set at k and f_{low} is set at $k-1$. The additively decomposable functions (ADFs), denoted by $F_{m\times k}$, are defined as:

$$F_{m\times k}(K_1 \dots K_m) = \sum_{i=1}^{m} F_k(K_i),\ K_i \in \{0,1\}^k. \tag{2}$$

The m and k are varied to produce a number of test functions. The ADFs fool gradient-based optimizers to favor zeroes, but the optimal solution is composed of all ones. The trap function is a fundamental unit for designing test functions that resist hill-climbing algorithms. The test functions can be effectively solved by composing BBs. Several discussions of the test functions can be found in [5, 11].

To illustrate the difficulty, the 10×5-trap functions ($F_{10\times5}$) is picked as an example. There are two different schemes to encode a solution to a binary string, $B = b_1 \dots b_{50}$ where $b_i \in \{0,1\}$ (see Figure 2). To compare the performance of a simple GA on both encoding schemes, we count the number of subfunctions (F_5) that are solved in the elitist individual observed during a run. The number of subfunctions that are solved is averaged from ten runs. The maximum number

of generations is set at 10,000. In Table 1, it can be seen that the first encoding scheme is better than the second encoding scheme. Both encoding schemes share the same optimal solution, but the first encoding scheme gives shorter defining-length BBs. Increasing the population size does not make the second encoding scheme better.

Encoding scheme 1:
$F_{10\times5}(B) = \sum_{i=1}^{10} F_5(b_{5i-4}b_{5i-3}b_{5i-2}b_{5i-1}b_{5i})$
Encoding scheme 1's BB: (defining length = 4)
11111***
Encoding scheme 2:
$F_{10\times5}(B) = \sum_{i=1}^{10} F_5(b_i b_{i+10}b_{i+20}b_{i+30}b_{i+40})$
Encoding scheme 2's BB: (defining length = 40)
1*********1*********1*********1*********1*********

Fig. 2. The performance of simple GA relies on the encoding scheme. An improper encoding scheme reduces the performance dramatically

Table 1. The performance of simple GA on the 10×5-trap function. The second encoding scheme gives long defining length, and therefore resulting in poor performance

Population size	Average subfunctions that are solved	
	Encoding scheme 1	Encoding scheme 2
100	4.0	0.8
1,000	8.4	0.7
10,000	10.0	0.1

Thierens raised the scalability issue of simple GAs [10]. He used the uniform crossover so that solutions are randomly mixed. The fitness function is the sum of 5-bit trap functions ($F_{m\times5}$). The analysis shows that either the computational time grows exponentially with the number of 5-bit trap functions or the population size must be exponentially increased. It is clear that scaling up the problem size requires BB information. In addition, the performance of simple GAs relies on the ordering of solution bits. The ordering may not pack the dependent bits close together. Such an ordering results in poor BB mixing. Therefore the BBs need to be identified to improve the scalability issue.

Many strategies in the literature use bit-reordering approach to pack the dependent bits close together, for example, inversion operator [2], messy GA [2], and linkage learning GA (LLGA) [3].

The inversion operator is shown in Figure 3. First, a chunk of adjacent bits are randomly selected. Next, the chunk is inversed by left-to-right flipping. The bits are moved around, but the meaning (fitness) of the solution does not changed. Only the ordering of solution bits is greatly affected. The bits at positions 4 and 7 are passed together with a higher probability. The inversion operator alters the ordering at random. The tight ordering (dependent bits being close

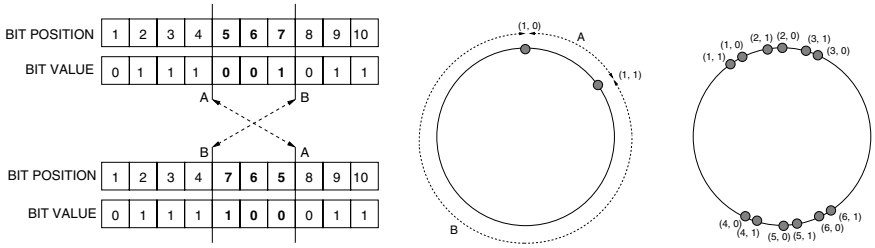


Fig. 3. Inversion operator (left) and linkage learning genetic algorithm (right)

together) are more likely to appear in the final generation. The simple GA enhanced with inversion operator is able to find the optimal solution for additively decomposable functions. In the worst case, the number of function evaluations grows exponentially with the problem size.

The messy GA encodes a solution bit to (p, v) where $p \in \{1, \ldots, \ell\}$ is bit position and $v \in \{0, 1\}$ is bit value. For instance, "01101" is encoded as (1, 0) (2, 1) (3, 1) (4, 0) (5, 1). The bits are tagged with the position numbers so that they can be moved around without losing the meaning. When the solution is mixed, the mixed solution may be over-specified or under-specified. The over-specification is having more than one copy for a bit position. The under-specification is having no copy for a bit position. Several alternatives are proposed for interpreting the over-specified and the under-specified solutions. For example, the over-specification is resolved by majority voting or first-come, first-serve basis. The under-specification is resolved by means of the competitive templates [2]. The messy GA is later developed to fast messy genetic algorithms (FMGA) and gene messy genetic algorithm (GEMGA) [2].

The LLGA encodes ℓ-bit solutions to 2ℓ distinct pieces of (p, v) placed on a circular string where the bit position $p \in \{1, \ldots, \ell\}$ and the bit value $v \in \{0, 1\}$. The 1-bit solution is encoded as it is shown in Figure 3 (left circle). Interpreting the solution is probabilistic. First, a starting point on the circular string is chosen. Second, walking clockwise and picking up (p, v) by first-come, first-serve basis. For instance, if $(1, 0)$ is encountered first, $(1, 1)$ will not be picked up. The 1-bit solution will be interpreted as $(1, 0)$ with probability $\frac{B}{A+B}$, but the interpretation will be $(1, 1)$ with probability $\frac{A}{A+B}$ where A and B are distances on the circular string. In Figure 3 (right circle), the dependent bits come close together. The solution will be interpreted as "111111" with a high probability.

The bit-reordering approach does not explicitly identify BBs, but it successfully delivers the optimal solution. Several papers explicitly analyze the fitness function. The analysis is done on a set of random solutions. Munetomo proposed that bit i and bit j should be in the same BBs if the monotonicity is violated by at least a solution [7]. The monotonicity is defined as follows.

if $\triangle f_i(s) > 0$ and $\triangle f_j(s) > 0$ then $\triangle f_{ij}(s) > \triangle f_i(s)$ and $\triangle f_{ij}(s) > \triangle f_j(s)$
if $\triangle f_i(s) < 0$ and $\triangle f_j(s) < 0$ then $\triangle f_{ij}(s) < \triangle f_i(s)$ and $\triangle f_{ij}(s) < \triangle f_j(s)$

where

$$\triangle f_i(s) = f(...\overline{s}_i...) - f(...s_i...) \tag{3}$$

$$\triangle f_{ij}(s) = f(..\overline{s}_i..\overline{s}_j..) - f(..s_i..s_j..) \quad (4)$$

f denotes fitness function. s denotes binary string. s_i is the i^{th} bit of s. $\overline{s}_i$ denotes $1 - s_i$. In practice, there might be a relaxation of the monotonicity condition.

Another work that explicitly identifies BBs is to reconstruct the fitness function [6]. Any function can be written in terms of Walsh's functions. For example, $f(x_1, x_2, x_3)$, can be written as $f(x_1, x_2, x_3) = w_0 + w_1\Psi_1(x_1) + w_2\Psi_2(x_2) + w_3\Psi_3(x_3) + w_4\Psi_4(x_1, x_2)\ w_5\Psi_5(x_1, x_3) + w_6\Psi_6(x_2, x_3) + w_7\Psi_7(x_1, x_2, x_3)$ where w_i is Walsh's coefficient ($w_i \in R$) and Ψ_i is Walsh's function ($\Psi_i : R \times \ldots \times R \rightarrow \{-1, 1\}$). The main algorithm is to compute the Walsh's coefficients. The non-zero Walsh's coefficient indicates the dependency between its associated variables. However, the number of Walsh's coefficients grows exponentially with variables. An underlying assumption is that the function has bounded variable interaction of order-k. Subsequently, the Walsh's coefficients can be calculated in a polynomial time.

Identifying BBs is somewhat related to building a distribution of solutions [4, 8]. The basic concept of optimization by building a distribution is to start with a uniform distribution of solutions. Next, a number of solutions is drawn from the distribution. Some good solutions (winners) are selected. Then the distribution is adjusted toward the winners (the winners-like solutions will be drawn with a higher probability in the next iteration). These steps are repeated until the optimal solution is found or reaching a termination condition. The work in this category is referred to as probabilistic model-building genetic algorithms (PMBGAs).

The Bayesian optimization algorithm (BOA) uses the Bayesian network to represent a distribution [9]. It is shown that if the problem is composed of k-bit trap functions, the network will be fully connected sets of k nodes (see Figure 4) [9, pp. 54]. In addition, the Bayesian network is able to represent joint distributions in the case of overlapped BBs. The BOA can solve the sum of k-bit trap functions ($F_{m\times k}$) in a polynomial relationship between the number of function evaluations and the problem size [9]. The hierarchical BOA (hBOA) is the BOA enhanced with decision tree/graph and a niching method called restricted tournament replacement [9]. The hBOA can solve the hierarchically decomposable functions (HDFs) in a scalable manner. Successful applications for BB identification are financial applications, distributed data mining, cluster optimization, maximum satisfiability of logic formulas (MAXSAT) and Ising spin glass systems [2].

We have present many techniques for identifying BBs. Those techniques have different strength and consume different computational time. The Bayesian network is a powerful tool for identifying BBs, but building the network is time-consuming. Eventually there will be a parallel construction of Bayesian networks. This paper presents a distinctive approach for identifying BBs. Let S be a set of binary solutions and the solution $s = b_1 \ldots b_\ell,\ b_i \in \{0, 1\}$. We construct a symmetric matrix of which the element in row i and column j, denoted by m_{ij}, is the chi-square of variables b_i and b_j. The matrix is called chi-square matrix (CSM). The CSM is further developed from our previous work, the simultaneity

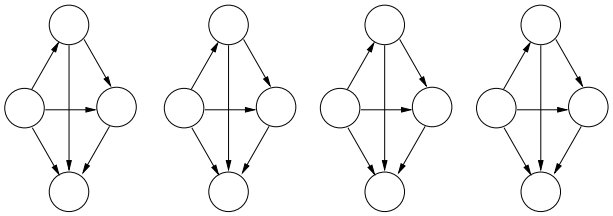


Fig. 4. A final structure of the Bayesian network. An edge indicates dependency between two variables

matrix (SM) [1]. The larger the m_{ij} is, the higher the dependency is between bit i and bit j. The matrix computation is simple and fast. Recently, there is similar work called dependency structure matrix (DSM) [12]. An element of the DSM is only zero (independent) or one (dependent) that is determined by the non-monotonicity [7]. Computing the CSM differs from that of the DSM. However, the papers that are independently developed share some ideas. The remainder of the paper is organized as follows. Section 2 describes the chi-square matrix. Section 3 validates the chi-square matrix with a number of test functions. Section 4 makes a comparison to the BOA and the hBOA. Section 5 concludes the paper.

2 The Chi-Square Matrix

Let $M = (m_{ij})$ be an $\ell \times \ell$ symmetric matrix of numbers. Let S be a set of ℓ-bit binary strings. Let s_i be the i^{th} string, $1 \leq i \leq n$. Let $s_i[j]$ be the j^{th} bit of s_i, $1 \leq j \leq \ell$. The chi-square matrix (CSM) is defined as follows.

$$m_{ij} = \begin{cases} ChiSquare(i,j) & ; \text{ if } i \neq j \\ 0 & ; \text{ otherwise.} \end{cases} \tag{5}$$

The $ChiSquare(i,j)$ is defined as:

$$\sum_{xy} \frac{(C_S^{xy}(i,j) - n/4)^2}{n/4}, \ (x,y) \in \{0,1\}^2 \tag{6}$$

where $\mathrm{C}_S^{xy}(i,j)$ counts the number of solutions in which the bit i and the bit j are "00," "01," "10," and "11." The expected frequency of observing "00" or "01" or "10" or "11" is $n/4$ where n is the number of solutions. If the solutions are random, the observed frequency $\mathrm{C}_S^{xy}(i,j)$ is close to the expected frequency. The common structures (or building-blocks) appear more often than the expected frequency. Consequently, the chi-square of bit variables that are in the same BB is high. The time complexity of computing the matrix is $O(\ell^2 n)$.

3 A Validation of the CSM

The building-block hypothesis states that the solution quality can be improved by composing BBs. The artificial functions are designed so that the BB hypothe-

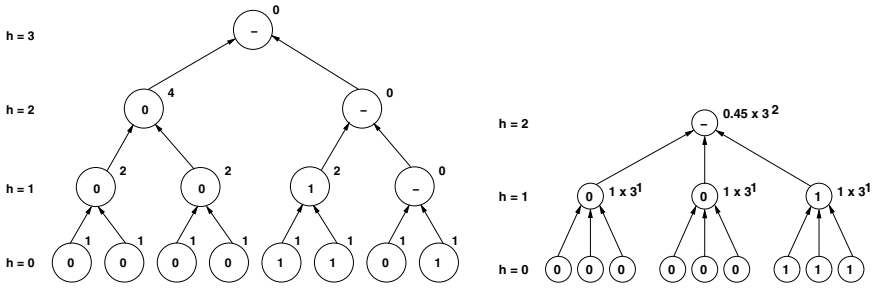


Fig. 5. The HIFF function interprets the solution as a binary tree (left). The 8-bit solution, "00001101," is placed at the lowest level ($h = 0$). The interpretation results are "0," "1," and "–" according to a deterministic rule. Each node excepting the nodes that are "–"contributes to the fitness by 2^h. The fitness is a total of 18. The HTrap1 function interprets the solution as a 3-branch tree (right). The 9-bit solution, "000000111," is placed at the lowest level ($h = 0$). The interpretation results are "0," "1," and "–" according to a deterministic rule. Each node excepting the leaf nodes contributes to the fitness by $3^h \times F_3(b_1 b_2 b_3)$ where b_i is the interpretation of the child nodes. The fitness of "000000111" is 13.05

sis is true, for example, the additively decomposable functions (ADFs) mentioned in the first section and the hierarchically decomposable functions (HDFs). The HDFs are far more difficult than the ADFs. First, BBs in the lowest level need to be identified. The solution quality is improved by exploiting the identified BBs in solution recombination. Next, the improved population reveals larger BBs. Again the BBs in higher levels need to be identified. Identifying and exploiting BBs are repeated many times until reaching the optimal solution. Commonly used HDFs are hierarchically if-and-only-if (HIFF), hierarchical trap 1 (HTrap1), and hierarchical trap 2 (HTrap2) functions. The original definitions of the HDFs can be found in [11, 9].

To compute the HIFF functions, a solution is interpreted as a binary tree. An example is shown in Figure 5 (left). The solution is an 8-bit string, "00001101." It is placed at the leaf nodes of the binary tree. The leaf nodes are interpreted as the higher levels of the tree. A pair of zeroes and a pair of ones are interpreted as zero and one respectively. Otherwise the interpretation result is "–." The HIFF functions return the sum of values contributed from each node. The contribution of node i, c_i, shown at the upper right of the node, is defined as:

$$c_i = \begin{cases} 2^h & ; \text{ if node } i \text{ is "0" or "1"} \\ 0 & ; \text{ if node } i \text{ is "–,"} \end{cases} \tag{7}$$

where h is the height of node i. In the example, the fitness of "00001101" is $\sum c_i = 18$. The HIFF functions do not bias an optimizer to favor zeroes rather than ones and vice versa. There are two optimal solutions, the string composed of all zeroes and the string composed of all ones.

The HTrap1 functions interpret a solution as a tree in which the number of branches is greater than two. An example is shown in Figure 5 (right). The

solution is a 9-bit string placed at the leaf nodes. The leaf nodes do not contribute to the function. The interpretation rule is similar to that of the HIFF functions. Triple zeroes are interpreted as zero and triple ones are interpreted as one. Otherwise the interpretation result is "–." The contribution of node i, c_i, is defined as:

$$c_i = \begin{cases} 3^h \times F_3(b_1 b_2 b_3) & ; \text{ if } b_j \neq \text{“–”} \text{ for all } 1 \leq j \leq 3 \\ 0 & ; \text{ otherwise,} \end{cases} \tag{8}$$

where h is the height of node i. b_1, b_2, b_3 are the interpretations in the left, middle, right children of node i. At the root node, the trap function's parameters are $f'_{high} = 1$ and $f'_{low} = 0.9$. The other nodes use $f_{high} = 1$ and $f_{low} = 1$. In Figure 5 (right), the HTrap1 function returns $\sum c_i = 13.05$. The optimal solution is composed of all ones.

The HTrap2 functions are similar to the HTrap1 functions. The only difference is the trap function's parameters. In the root node, $f'_{high} = 1$ and $f'_{low} = 0.9$. The other nodes use $f_{high} = 1$ and $f_{low} = 1 + \frac{0.1}{h}$ where h is tree height. The optimal solution is composed of all ones if the following condition is true.

$$f'_{high} - f'_{low} > (h-1)(f_{low} - f_{high}) \tag{9}$$

The parameter setting ($f'_{high} = 1$, $f'_{low} = 0.9$, $f_{high} = 1$, $f_{low} = 1 + \frac{0.1}{h}$) satisfies the condition. The HTrap2 functions are more deceptive than the HTrap1 functions. Only root node guides an optimizer to favor ones while the other nodes fool the optimizer to favor zeroes by setting $f_{low} > f_{high}$.

To validate the chi-square matrix, an experiment is set as follows. We randomize a population of which the fitness of any individual is greater than a threshold T. Next, the matrix is computed according to the population. Every time step, the threshold T is increased and the matrix is recomputed. The population size is set at 50 for all test functions. The onemax function counts the number of ones. The mixed-trap function is additively composed of 5-bit onemax, 3-bit, 4-bit, 5-bit, 6-bit, and 7-bit trap functions. A sequence of the chi-square matrix is shown in Figure 6-7. A matrix element is represented by a square. The square intensity is proportional to the value of matrix element In the early stage (A), the population is almost random because the threshold T is small. Therefore there are no irregularities in the matrix. The solution quality could be slightly improved without the BB information. That is sufficient to reveal some irregularities or BBs in the next population (B). Improving the solution quality further requires information about BBs (C). Otherwise, randomly mixing disrupts the BBs with a high probability. Finally, the population contains only high-quality solutions. The BBs are clearly seen (D). The correctness of the BBs depends on the quality of solutions observed. An optimization algorithm that exploits the matrix have to extract the BB information from the matrix in order to perform the solution recombination. Hence, moving the population from (A) to (B), (B) to (C), and (C) to (D).

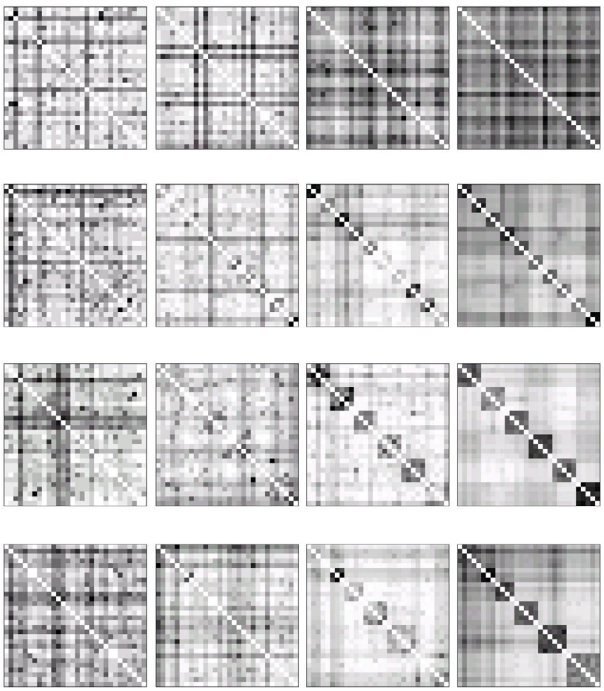


Fig. 6. The chi-square matrix constructed from a set of 50 random solutions. The fitness of any individual in the population is greater than the threshold T. The ADFs have only single-level BBs

4 A Comparison to BOA and hBOA

An exploitation of the chi-square matrix is to compute a partition $\{1, \ldots, \ell\}$ where ℓ is the solution length [1]. The main idea is to put i and j in the same partition subset if the matrix element m_{ij} is significantly high. There are several definitions of the desired partition, for example, the definitions in the senses of non-monotonicity [7], Walsh coefficients [6], and minimal description-length principle [4]. We develop a definition in the sense of chi-square matrix. Algorithm PAR searches for a partition P such that

1. P is a partition.
   1.1 The members of P are disjoint set.
   1.2 The union of all members of P is $\{1, \ldots, \ell\}$.
2. $P \neq \{\{1, \ldots, \ell\}\}$.
3. For all $B \in P$ such that $|B| > 1$,
   3.1 for all $i \in B$, the largest $|B| - 1$ matrix elements in row i are founded in columns of $B \setminus \{i\}$.
4. For all $B \in P$ such that $|B| > 1$,
   4.1 $H_{max} - H_{min} < \alpha(H_{max} - L_{min})$ where

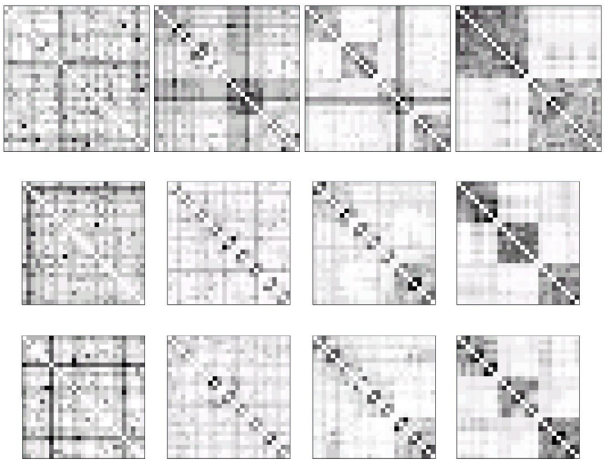
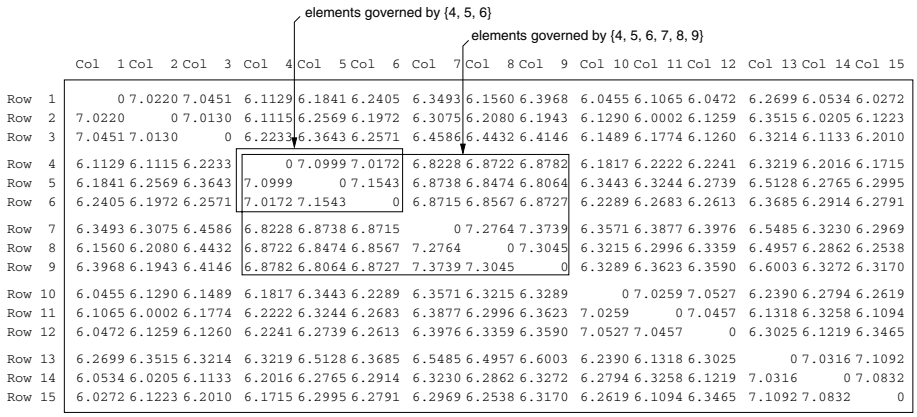


Fig. 7. The chi-square matrix constructed from a set of 50 random solutions. The fitness of any individual in the population is greater than the threshold T. The HDFs have multiple-level BBs

$H_{max} = max(\{m_{ij} \mid (i,j) \in B^2,\ i \neq j\})$,
$H_{min} = min(\{m_{ij} \mid (i,j) \in B^2,\ i \neq j\})$,
$L_{min} = min(\{m_{ij} \mid i \in B, j \in \{1,\ldots,\ell\} \setminus B\})$, and $\alpha \in [0,1]$.

5. There are no partition P_x such that for some $B \in P$, for some $B_x \in P_x$, P and P_x satisfy the first, the second, the third, and the fourth conditions, $B \subset B_x$.

We assume that the matrix elements $\{m_{ij} \mid i < j\}$ are distinct. In practice, the elements can be made distinct by several techniques [1]. An example of the chi-square matrix is shown in Figure 8. The first condition is obvious. The second condition does not allow the coarsest partition because it is not useful in solution recombination. The third condition makes i and j, in which m_{ij} is significantly high, in the same partition subset. For instance, $P_1 = \{\{1,2,3\}, \{4,5,6\}, \{7,8,9\}, \{10,11,12\}, \{13,14,15\}\}$ satisfies the third condition because the largest two elements in row 1 are found in columns of $\{2,3\}$, the largest two elements in row 2 are found in columns of $\{1,3\}$, the largest two elements in row 3 are found in columns of $\{1,2\}$, and so on. However, there are many partitions that satisfy the third condition, for example, $P_2 = \{\{1,2,3\}, \{4,5,6,7,8,9\}, \{10,11,12\}, \{13,14,15\}\}$. There is a dilemma between choosing the fine partition (P_1) and the coarse partition (P_2). Choosing the fine partition prevents the emergence of large BBs, while the coarse partition results in poor mixing. To overcome the dilemma, the coarse partition will be acceptable if it satisfies the fourth condition. The fifth condition says choosing the coarsest partition that is consistent with the first, the second, the third, and the fourth conditions.



	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8	Col 9	Col 10	Col 11	Col 12	Col 13	Col 14	Col 15
Row 1	0	7.0220	7.0451	6.1129	6.1841	6.2405	6.3493	6.1560	6.3968	6.0455	6.1065	6.0472	6.2699	6.0534	6.0272
Row 2	7.0220	0	7.0130	6.1115	6.2569	6.1972	6.3075	6.2080	6.1943	6.1290	6.0002	6.1259	6.3515	6.0205	6.1223
Row 3	7.0451	7.0130	0	6.2233	6.3643	6.2571	6.4586	6.4432	6.4146	6.1489	6.1774	6.1260	6.3214	6.1133	6.2010
Row 4	6.1129	6.1115	6.2233	0	7.0999	7.0172	6.8228	6.8722	6.8782	6.1817	6.2222	6.2241	6.3219	6.2016	6.1715
Row 5	6.1841	6.2569	6.3643	7.0999	0	7.1543	6.8738	6.8474	6.8064	6.3443	6.3244	6.2739	6.5128	6.2765	6.2995
Row 6	6.2405	6.1972	6.2571	7.0172	7.1543	0	6.8715	6.8567	6.8727	6.2289	6.2683	6.2613	6.3685	6.2914	6.2791
Row 7	6.3493	6.3075	6.4586	6.8228	6.8738	6.8715	0	7.2764	7.3739	6.3571	6.3877	6.3976	6.5485	6.3230	6.2969
Row 8	6.1560	6.2080	6.4432	6.8722	6.8474	6.8567	7.2764	0	7.3045	6.3215	6.2996	6.3359	6.4957	6.2862	6.2538
Row 9	6.3968	6.1943	6.4146	6.8782	6.8064	6.8727	7.3739	7.3045	0	6.3289	6.3623	6.3590	6.6003	6.3272	6.3170
Row 10	6.0455	6.1290	6.1489	6.1817	6.3443	6.2289	6.3571	6.3215	6.3289	0	7.0259	7.0527	6.2390	6.2794	6.2619
Row 11	6.1065	6.0002	6.1774	6.2222	6.3244	6.2683	6.3877	6.2996	6.3623	7.0259	0	7.0457	6.1318	6.3258	6.1094
Row 12	6.0472	6.1259	6.1260	6.2241	6.2739	6.2613	6.3976	6.3359	6.3590	7.0527	7.0457	0	6.3025	6.1219	6.3465
Row 13	6.2699	6.3515	6.3214	6.3219	6.5128	6.3685	6.5485	6.4957	6.6003	6.2390	6.1318	6.3025	0	7.0316	7.1092
Row 14	6.0534	6.0205	6.1133	6.2016	6.2765	6.2914	6.3230	6.2862	6.3272	6.2794	6.3258	6.1219	7.0316	0	7.0832
Row 15	6.0272	6.1223	6.2010	6.1715	6.2995	6.2791	6.2969	6.2538	6.3170	6.2619	6.1094	6.3465	7.1092	7.0832	0

Fig. 8. An example of the chi-square matrix. The matrix elements in the diagonal are always zero. The matrix is symmetric ($m_{ij} = m_{ji}$)

By condition 4.1, the partition subset $\{4, 5, 6\}$ is acceptable because the values of matrix elements governed by $\{4, 5, 6\}$ are close together (see Figure 8). Being close together is defined by $H_{max} - H_{min}$ where H_{max} and H_{min} is the maximum and the minimum of the nondiagonal matrix elements governed by the partition subset. The $H_{max} - H_{min}$ is a degree of irregularities of the matrix. The main idea is to limit $H_{max} - H_{min}$ to a threshold. The threshold, $\alpha(H_{max} - L_{min})$, is defined relatively to the matrix elements because the threshold cannot be fixed for a problem instance. The partition subset $\{4, 5, 6\}$ gives $H_{max} = 7.1543$, $H_{min} = 7.0172$, and $L_{min} = 6.1115$. L_{min} is the minimum of the nondiagonal matrix elements in rows of $\{4, 5, 6\}$. The fourth condition limits $H_{max} - H_{min}$ to $100 \times \alpha$ percent of the difference between H_{max} and L_{min}. An empirical study showed that α should be set at 0.90 for both ADFs and HDFs. Choosing $\{4, 5, 6, 7, 8, 9\}$ yields $(H_{max} = 7.3739, H_{min} = 6.8064, L_{min} = 6.1115)$ which does not violate condition 4.1. The fifth condition prefers a coarse partition $\{\{4, 5, 6, 7, 8, 9\}, \ldots\}$ to a fine partition $\{\{4, 5, 6\}, \ldots\}$ so that the partition subsets can be grown to compose larger BBs in higher levels.

Algorithm PAR is shown in Figure 9. A trace of the algorithm is shown in Table 2. The outer loop processes row 1 to ℓ. In the first step, the columns of the sorted values in row i are stored in $R_{i,1}$ to $R_{i,\ell}$. For $i = 1$, $R_{1,1}$ to $R_{i,\ell}$ are 3, 2, 9, 7, 13, 6, 5, 8, 4, 11, 14, 12, 10, 15, 1, respectively. Next, the inner loop tries a number of partition subsets by enlarging A ($A \leftarrow A \cup \{R_{i,j}\}$). If A satisfies conditions 3.1 and 4.1, A will be saved to B. Finally, P is the partition that satisfies the five conditions. Checking conditions 3.1 and 4.1 is the most time-consuming section. It can be done in $O(\ell^2)$. The checking is done at most ℓ^2 times. Therefore the time complexity of PAR is $O(\ell^4)$.

We customize simple GAs as follows. Every generation, the chi-square matrix is constructed. The PAR algorithm is executed to find the partition. Two parents are chosen by the roulette-wheel method. The solutions are reproduced by a restricted uniform crossover – bits governed by the same partition subset must

```
M = (m_ij) denotes ℓ×ℓ chi-square matrix.
T_i and R_{i,j} denote arrays of numbers.
A and B are partition subsets.
P denotes a partition.

Algorithm PAR(M, α)
P ← ∅;
for i = 1 to ℓ do    // outer loop
    if i ∉ B for all B ∈ P then
        T ← {row i sorted in desc. order};
        for j = 1 to ℓ do
            R_{i,j} ← x where m_ix = T_j;
        endfor
        A ← {i};
        B ← {i};
        for j = 1 to ℓ − 2 do    // inner loop
            A ← A ∪ {R_{i,j}};
            if A satisfies cond. 3.1 and 4.1 then
                B ← A;
        endfor
        P ← P ∪ {B};
    endif
endfor
return P;
```

Fig. 9. Algorithm PAR takes an $\ell \times \ell$ symmetric matrix, $M = (m_{ij})$. The output is the partition of $\{1, \ldots, \ell\}$

be passed together. The mutation is turned off. The diversity is maintained by the rank-space method. The population size is determined empirically by the bisection method [9, pp. 64]. The bisection method performs binary search for the minimal population size. There might be 10% different between the population size used in the experiments and the minimal population size that ensures the optimal solution in all independent 10 runs.

The chi-square matrix (CSM) is compared to the Bayesian optimization algorithm (BOA) [9, pp. 115–117]. We also show the results of our previous work, the simultaneity matrix (SM) [1]. Figure 10 shows the number of function evaluations required to reach the optimum. The linear regression in log scale indicates a polynomial relationship between the number of function evaluations and the problem size. The degree of polynomial can be approximated by the slope of linear regression. The maximum number of incoming edges, a parameter of the BOA, limits the number of incoming edges for every vertex in the Bayesian network. The default setting is to set the number of incoming edges to $k - 1$ for $m \times k$-trap functions. It can be seen that the BOA and the CSM can solve the ADFs in a polynomial relationship between the number of function evaluations and the problem size. The BOA performs better than the CSM. However, the performance gap narrows as the problem becomes harder (onemax, $m \times 3$-trap,

Table 2. A trace of the PAR algorithm is shown in the table below. The PAR input is the matrix in Figure 8. The partition subset A is enlarged by $R_{i,j}$. If A satisfies conditions 3.1 and 4.1, A will be saved to B. After finishing the iteration $i = 1$ and $j = 13$, B is added to the partition. Finally, PAR returns the partition $\{\{1,2,3\},\{4,5,6,7,8,9\},\{10,11,12\},\{13,14,15\}\}$

i	j	A	Cond. 3.1	Cond. 4.1	B
1	1	{1, 3}	True	True	{1, 3}
1	2	{1, 3, 2}	True	True	{1, 3, 2}
1	3	{1, 3, 2, 9}	False	False	{1, 3, 2}
1	4	{1, 3, 2, 9, 7}	False	False	{1, 3, 2}
1	5	{1, 3, 2, 9, 7, 13}	False	False	{1, 3, 2}
1	6	{1, 3, 2, 9, 7, 13, 6}	False	False	{1, 3, 2}
1	7	{1, 3, 2, 9, 7, 13, 6, 5}	False	False	{1, 3, 2}
1	8	{1, 3, 2, 9, 7, 13, 6, 5, 8}	False	False	{1, 3, 2}
1	9	{1, 3, 2, 9, 7, 13, 6, 5, 8, 4}	False	False	{1, 3, 2}
1	10	{1, 3, 2, 9, 7, 13, 6, 5, 8, 4, 11}	False	False	{1, 3, 2}
1	11	{1, 3, 2, 9, 7, 13, 6, 5, 8, 4, 11, 14}	False	False	{1, 3, 2}
1	12	{1, 3, 2, 9, 7, 13, 6, 5, 8, 4, 11, 14, 12}	False	False	{1, 3, 2}
1	13	{1, 3, 2, 9, 7, 13, 6, 5, 8, 4, 11, 14, 12, 10}	False	False	{1, 3, 2}

and $m\times5$-trap functions respectively). The difficulty of predetermining the maximum number of incoming edges is resolved in a later version of the BOA called hierarchical BOA (hBOA). We made a comparison between the CSM and the hBOA [9, pp. 164–165] (see Figure 10). The hBOA uses less number of function evaluations than that of the CSM. But in terms of scalability, the hBOA and the CSM are able to solve the HDFs in a polynomial relationship.

Another comparison to the hBOA is made in terms of elapsed time and memory usage. The elapsed time is an execution time of a call on subroutine `constructTheNetwork`. The memory usage is the number of bytes dynamically allocated in the subroutine. The hardware platform is HP NetServer E800, 1GHz Pentium-III, 2GB RAM, and Windows XP. The memory usage in the hBOA is very large because of inefficient memory management in constructing the Bayesian network. A fair implementation of the Bayesian network is the Microsoft WinMine Toolkit. The WinMine is a set of tools that allow you to build statistical models from data. It constructs the Bayesian network with decision tree that is similar to that of the hBOA. The WinMine's elapsed time and memory usage are measured by an execution of `dnet.exe` – a part of the WinMine that constructs the network. All experiments are done with the same biased population that is composed of aligned chunks of zeroes and ones. The parameters of the hBOA and the WinMine Toolkit are set at default. The population size is set at three times greater than the problem size. The elapsed time and memory usage averaged from 10 independent runs are shown in Figure 11. It can be seen that constructing the Bayesian network is time-consuming. In contrast, the matrix computation is 10 times faster and uses 10 times less memory then constructing the network.

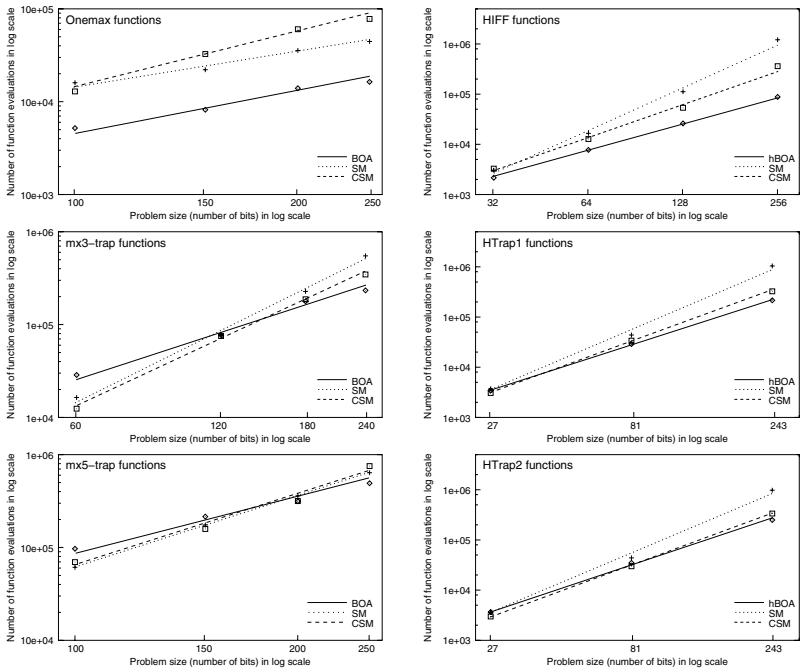


Fig. 10. Performance comparison between the BOA and the CSM on ADFs (left) Performance comparison between the hBOA and the CSM on HDFs (right)

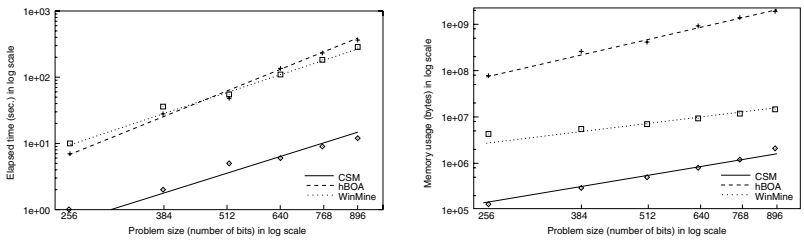


Fig. 11. Elapsed time (left) and memory usage (right) required to construct the Bayesian network and the upper triangle of the matrix (a half of the matrix is needed because the matrix is symmetric)

5 Conclusions

The current BB identification research relies on building a distribution of solutions. The Bayesian network is a powerful representation for the distribution. Nevertheless, building the network is time-and-memory consuming. We have presented a BB identification by the chi-square matrix. The matrix element m_{ij} is the degree of dependency between bit i and bit j. The time complexity of com-

puting the matrix is $O(\ell^2 n)$ where ℓ is the solution length and n is the number of solutions. We put i and j of which m_{ij} is high in the same partition subset. The time complexity of partitioning is $O(\ell^4)$ where ℓ is the solution length. The bits governed by the same partition subsets are passed together when performing solution recombination. The chi-square matrix is able to solve the ADFs and HDFs in a scalable manner. In addition, the matrix computation is efficient in terms of computational time and memory usage.

References

1. Aporntewan, C., and Chongstitvatana, P. (2004). Simultaneity matrix for solving hierarchically decomposable functions. Proceedings of the Genetic and Evolutionary Computation, page 877–888, Springer-Verlag, Heidelberg, Berlin.
2. Goldberg, D. E. (2002). The Design of Innovation: Lessons from and for Competent Genetic Algorithms. Kluwer Academic Publishers, Boston, MA.
3. Harik, G. R. (1997). Learning linkage. Foundation of Genetic Algorithms 4, page 247–262, Morgan Kaufmann, San Francisco, CA.
4. Harik, G. R. (1999). Linkage learning via probabilistic modeling in the ECGA. Technical Report 99010, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Champaign, IL.
5. Holland, J. H. (2000). Building blocks, cohort genetic algorithms, and hyperplane-defined functions. Evolutionary Computation, Vol. 8, No. 4, page 373–391, MIT Press, Cambridge, MA.
6. Kargupta, H., and Park, B. (2001). Gene expression and fast construction of distributed evolutionary representation. Evolutionary Computation, Vol. 9, No. 1, page 43–69, MIT Press, Cambridge, MA.
7. Munetomo, M., and Goldberg, D. E. (1999). Linkage identification by non-monotonicity detection for overlapping functions. Evolutionary Computation, Vol. 7, No. 4, page 377–398, MIT Press, Cambridge, MA.
8. Pelikan, M., Goldberg, D. E., and Lobo, F. (1999). A survey of optimization by building and using probabilistic models. Computational Optimization and Applications, Vol. 21, No. 1, page 5–20, Kluwer Academic Publishers.
9. Pelikan, M. (2002). Bayesian optimization algorithm: From single level to hierarchy. Doctoral dissertation, University of Illinois at Urbana-Champaign, Champaign, IL.
10. Thierens, D. (1999). Scalability problems of simple genetic algorithms. Evolutionary Computation, Vol. 7, No. 4, page 331–352, MIT Press, Cambridge, MA.
11. Watson, R. A., and Pollack, J. B. (1999). Hierarchically consistent test problems for genetic algorithms. Proceedings of Congress on Evolutionary Computation, page 1406–1413, IEEE Press, Piscataway, NJ.
12. Yu, T., and Goldberg, D. E. (2004). Dependency structure matrix analysis: off-line utility of the dependency structure matrix genetic algorithm. Proceedings of the Genetic and Evolutionary Computation, page 355–366, Springer-Verlag, Heidelberg, Berlin.