Soft Comput (2006)
DOI 10.1007/s00500-006-0097-z

ORIGINAL PAPER

Chatchawit Aporntewan · Prabhas Chongstitvatana

Building-block identification by simultaneity matrix

© Springer-Verlag 2006

Abstract This paper presents a study of building blocks (BBs) in the context of genetic algorithms (GAs). In GAs literature, the BBs are common structures of high-quality solutions. The aim is to identify and maintain the BBs while performing solution recombination. To identify the BBs, we construct an $\ell \times \ell$ simultaneity matrix according to a set of ℓ-bit solutions. The matrix element in row i and column j denoted by m_{ij} is the degree of dependency between bit i and bit j. We search for a partition of $\{0, \ldots, \ell - 1\}$ for the matrix. The main idea of partitioning is to put i and j of which m_{ij} is significantly high in the same partition subset. The partition represents the bit positions of BBs. The partition is exploited in solution recombination so that the bits governed by the same partition subset are passed together. It can be shown that by exploiting the simultaneity matrix the additively decomposable functions can be solved in a polynomial relationship between the number of function evaluations required to reach the optimum and the problem size. A comparison to the Bayesian optimization algorithm (BOA) is made. Empirical results show that the BOA uses less number of function evaluations than that of our algorithm. However, computing the matrix is ten times faster than constructing the Bayesian network.

1 Introduction

This paper presents a line of research in genetic algorithms (GAs), called building-block identification. The GAs are probabilistic search and optimization algorithm [7,13]. 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 and also disrupts the BBs because the cut point is chosen at random (see Fig. 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.

For some conditions [9, Chaps. 7–11], the success of GAs can be explained by the schema theorem and the building-block hypothesis [7,13]. The schema theorem states that the number of solutions that match the above average, short defining-length, and low-order schemata grows exponentially. The optimal solution is hypothesized to be composed of the above average schemata or the BBs. 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. When the good BBs are more disrupted, it is said to be a GA-hard problem. Trap function [1] is an adversary function for studying BBs and linkage problems in GAs [10]. The general k-bit trap functions are defined as:

$$F_k(b_0 \ldots b_{k-1}) = \begin{cases} f_{\text{high}}; & \text{if } u = k \\ f_{\text{low}} - u\frac{f_{\text{low}}}{k-1}; & \text{otherwise,} \end{cases} \quad (1)$$

where $b_i \in \{0, 1\}$, $u = \sum_{i=0}^{k-1} b_i$, 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:

C. Aporntewan (✉)
Department of Mathematics, Faculty of Science,
Chulalongkorn University, Bangkok 10330, Thailand
E-mail: Chatchawit.A@chula.ac.th

P. Chongstitvatana
Department of Computer Engineering, Faculty of Engineering,
Chulalongkorn University, Bangkok 10330, Thailand
E-mail: Prabhas.C@chula.ac.th

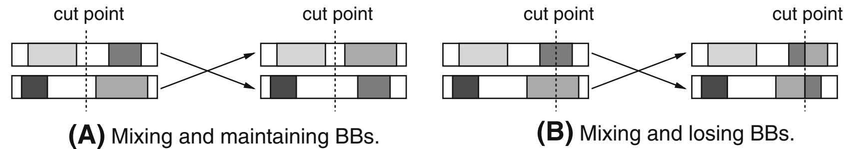

Fig. 1 The solutions are mixed by the crossover operator. The BBs are shadowed. The cut point, chosen at random, divides a solution into two pieces. Then, the pieces of solutions are exchanged. In case (**a**), the solutions are mixed while maintaining the BBs. In case (**b**), the BBs are disrupted

Table 1 A population of highly fit individuals

Individual no.	$b_0b_1b_2$	$b_3b_4b_5$	$b_6b_7b_8$	$b_9b_{10}b_{11}$	$b_{12}b_{13}b_{14}$	Fitness
1	111	111	000	111	000	13.0
2	000	000	111	000	111	12.0
3	111	000	000	111	000	12.0
4	000	000	000	000	111	11.0
5	000	000	000	000	000	10.0

The fitness is the sum of five three-bit trap functions. "111" is the optimum for three-bit trap function. "000" gives more contribution to the fitness than that of "001," "010," "011," "100," "101," and "110." As a result, the highly fit population is composed of "000" and "111"

$$F_{m\times k}(K_0 \ldots K_{m-1}) = \sum_{i=0}^{m-1} F_k(K_i),\ K_i \in \{0, 1\}^k. \tag{2}$$

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. 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 [6,12,14,32,33].

The BBs are inferred from a population of highly fit individuals [9, pp. 60–61]. A highly fit population is shown in Table 1. The fitness function is the sum of five three-bit trap functions. The dependency between variables b_i, b_{i+1}, b_{i+2} ($i = 0, 3, 6, 9, 12$) can be detected by means of a statistical method. An inference might be that the highly fit individuals are composed of triple zeroes and triple ones. It is said that the triple zeroes and triple ones are common traits or BBs. We aim to identify these BBs so that the BBs are maintained in solution recombination. Consequently, the optimal solution can be achieved by composing BBs.

Thierens raised the scalability issue of simple GAs [31]. He used the uniform crossover so that the solutions are randomly mixed. The fitness function is the sum of five-bit trap functions. The analysis shows that either the computational time grows exponentially with the number of five-bit trap functions or the population size must be exponentially increased. It is clear that scaling up the problem size requires information about the BBs so that the solutions are efficiently mixed. 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 the bit-reordering approach to pack the dependent bits close together, for example, inversion operator [7], messy GAs [8], symbiotic evolution [21], recombination strategy adaptation [30], adaptive linkage crossover [28], and linkage learning [10]. The bit-reordering approach does not explicitly identify BBs, but it successfully delivers the optimal solution. Several papers explicitly identify BBs. An approach is to find a partition of bit positions. For instance, Table 1 infers the partition:

$$\{\{0, 1, 2\}, \{3, 4, 5\}, \{6, 7, 8\}, \{9, 10, 11\}, \{12, 13, 14\}\}. \tag{3}$$

In the case of nonoverlapped BBs, partition is a clear representation [2,3,11,15,18,19]. Note that Kargupta [18] computes Walsh's coefficients which imply the partition. The bits governed by the same partition subset are passed together to prevent BB disruption.

Identifying BBs is somewhat related to building a distribution of solutions [4,5,11,20,23,24]. 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 according to the distribution. Some good solutions (winners) are selected, and the distribution is adjusted toward the winners (the winners-like solutions will be drawn with 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). For a particular form of distribution used in the extended compact genetic algorithm (ECGA), building the distribution is identical to searching for a partition [11]. The Bayesian optimization algorithm (BOA) uses Bayesian network to represent a distribution [23]. Pelikan showed that if the problem is composed of k-bit trap functions, the network will be fully connected sets of k nodes [26, pp. 54]. In addition, Bayesian network is able to represent joint distributions in the case of overlapping BBs. The hierarchical BOA (hBOA) is the BOA enhanced with decision tree/graph and a niching method called restricted tournament replacement [26]. The hBOA can solve the hierarchically decomposable functions (HDFs) in a scalable manner. Successful applications

for BB identification are financial applications [16], distributed data mining [17], cluster optimization [29], maximum satisfiability of logic formulas (MAXSAT) and Ising spin glass systems [27].

The Bayesian network is able to identify common structures in a population. Nevertheless, building the network is time-consuming. This paper presents a BB identification algorithm that is simpler and faster than that of the BOA. The algorithm is named building-block identification by simultaneity matrix (BISM). The BISM input is a set of ℓ-bit solutions. The BISM output is a partition of $\{0, \ldots, \ell-1\}$. Algorithm BISM consists of two parts: simultaneity matrix construction (SMC) and partitioning (PAR) algorithms. The SMC constructs the matrix according to a set of solutions. Next, PAR searches for a partition for the matrix. The remainder of the paper is organized as follows. Section 2 describes the SMC algorithm. Section 3 describes the PAR algorithm. Section 4 presents the experimental results and a comparison to the BOA. Section 5 concludes the paper.

2 Simultaneity matrix construction algorithm

The SMC input is a set of ℓ-bit binary string denoted by:

$$S = \{s_0, \ldots, s_{n-1}\}, \tag{4}$$

where s_i is the ith string, $0 \le i \le n-1$. The $s_i[j]$ denotes the jth bit of s_i, $0 \le j \le \ell - 1$. Algorithm SMC outputs an $\ell \times \ell$ symmetric matrix of numbers, denoted by $M = (m_{ij})$, $0 \le i, j \le \ell - 1$. The symmetric matrix is made for the simplicity of writing definitions. In practice, a half of the matrix is needed. A closed form of m_{ij} is shown in Eq. (5).

$$m_{ij} = \begin{cases} 0; & \text{if } i = j \\ \text{Count}_S^{00}(i,j) \times \text{Count}_S^{11}(i,j) & \\ \quad + \text{Count}_S^{01}(i,j) \times \text{Count}_S^{10}(i,j); & \text{otherwise,} \end{cases} \tag{5}$$

where $\text{Count}_S^{ab}(i,j) = |\{x \in \{0, \ldots, n-1\} : s_x[i] = a$ and $s_x[j] = b\}|$ for all $0 \le i, j \le \ell - 1$, $(a, b) \in \{0, 1\}^2$.

Algorithm SMC is shown in Fig. 2. Step 1 constructs only the upper triangle of the matrix using Eq. (5). Step 2 perturbs the matrix so that there are no identical elements. The matrix of which the elements are distinct is greatly helpful in partitioning. The perturbation does not totally change the matrix because each element is incremented by a small real random number ranging between 0 and 1. The perturbation by adding an integer with a real number is practical for most programming languages because it is hardly possible to produce identical random numbers. Step 3 copies the upper triangle $\{m_{ij} \mid i < j\}$ to the lower triangle $\{m_{ij} \mid i > j\}$. Step 4 returns the simultaneity matrix $M = (m_{ij})$. The time complexity of SMC is $O(\ell^2 n)$.

A matrix element m_{ij} is proportional to the probability that two-bit BBs at bit positions i and j will be disrupted by the uniform crossover. Considering all cases of mixing two-bit BBs. Mixing "00" with "11" results in "01" and "10." Mixing "01" with "10" results in "00" and "11." Only mixing in the two cases must be done carefully because the processing BBs will be lost. Mixing two-bit BBs in the other cases gives the same BBs. Therefore SMC algorithm counts a pair of two-bit BBs that are complement to each other. To exploit the matrix, the bits at positions i and j are passed together every time performing crossover if the matrix element m_{ij} is significantly high. The three-bit BBs are identified by inserting k to $\{i, j\}$. If the matrix elements m_{ij}, m_{jk}, and m_{ik} are significantly high, i, j, k should be in the same partition subset. Larger BBs can be identified in a similar fashion.

Trap functions embedded in the ADFs bias the population to two aligned chunks of zeroes and ones, that are complementary to each other. Certainly, the dependency between every pair of bits in the chunks is stored in the matrix. The matrix is not limited to the cases where the two aligned chunks are complement to each other. In the other cases, the matrix does not detect unnecessary dependency. For instance, the bits at positions of $\{0, 1, 2, 3, 4\}$ are mostly "b_0b_1000" and "b_0b_1111" where $b_i \in \{0, 1\}$. The dependency among five bits is obvious, but passing the bits governed by $\{2, 3, 4\}$ together is sufficient to guarantee that "b_0b_1000" and "b_0b_1111" will exist in the next generation with a high probability. In summary, the matrix records only dependency that is actually necessary for preserving BBs.

3 Partitioning algorithm

The PAR input is an $\ell \times \ell$ simultaneity matrix. The PAR outputs the partition:

$$P = \{B_0, \ldots, B_{|P|-1}\},$$

$$\bigcup_{i=0}^{|P|-1} B_i = \{0, \ldots, \ell - 1\}, \tag{6}$$

$$B_i \cap B_j = \emptyset \quad \text{for all } i \ne j.$$

The B_i is called partition subset. There are several definitions of the desired partition, for example, the definitions in the senses of nonmonotonicity [19], GEMGA [15], Walsh coefficients [18], and minimal description-length principle [11]. We develop a definition in the sense of simultaneity 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 $\{0, \ldots, \ell - 1\}$.
2. $P \ne \{\{0, \ldots, \ell - 1\}\}$.
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 $B \in P$,
   4.1 $|B| \le k$ where k is a predefined constant.
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, $B \subset B_x$.

An example of the simultaneity matrix is shown in Fig. 4. The perturbation is omitted because the values of $\{m_{ij} \mid i < j\}$

```
Algorithm SMC(S)
1. for i = 0 to ℓ − 1 do
      m_ii ← 0;
      for j = i + 1 to ℓ − 1 do
         m_ij ← Count_S^00(i, j) × Count_S^11(i, j) + Count_S^01(i, j) × Count_S^10(i, j);
2. for i = 0 to ℓ − 1 do
      for j = i + 1 to ℓ − 1 do
         m_ij ← m_ij + Random(0, 1);
3. for i = 0 to ℓ − 1 do
      for j = i + 1 to ℓ − 1 do
         m_ji ← m_ij;
4. return M = (m_ij);
```

Fig. 2 Algorithm SMC takes a set of ℓ-bit solutions, S, as input. In step 1, only the upper triangle of the matrix, $\{m_{ij} \mid i < j\}$, is constructed. Step 2 perturbs the matrix by adding small real random numbers to the matrix elements. Step 3 copies the upper triangle to the lower triangle of the matrix. Step 4 returns $\ell \times \ell$ symmetric matrix

```
Note: M = (m_ij) denotes ℓ×ℓ simultaneity matrix, 0 ≤ i, j ≤ ℓ − 1.
k is the maximum size of a partition subset.
T[i] and R[i] denote arrays of numbers indexed by 0 ≤ i ≤ ℓ − 1.
A and B are partition subsets. P denotes a partition.

Algorithm PAR(M, k)
P ← ∅;
for i = 0 to ℓ − 1 do    // outer loop processing row i
   if i ∉ B for all B ∈ P then
      array T[ ] = {matrix elements in row i sorted in descending order};
      for j = 0 to ℓ − 1 do R[j] = x where m_ix = T[j];
      A ← {i}; B ← {i};
      for j = 0 to k − 2 do    // inner loop enlarging A
         A ← A ∪ {R[j]};
         if A satisfies condition 3.1 then
            B ← A;
      endfor
      P ← P ∪ {B};
   endif
endfor
return P;
```

Fig. 3 Algorithm PAR takes an $\ell \times \ell$ symmetric matrix, $M = (m_{ij})$. The matrix elements $\{m_{ij} \mid i < j\}$ are distinct. The output is the partition of $\{0, \ldots, \ell - 1\}$

are distinct. 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, $P_1 = \{\{0, 1, 2\}, \{3, 4, 5\}, \{6, 7, 8\}, \{9, 10, 11\}, \{12, 13, 14\}\}$ satisfies the third condition because the largest two elements in row 0 are found in columns of $\{1, 2\}$, the largest two elements in row 1 are found in columns of $\{0, 2\}$, the largest two elements in row 2 are found in columns of $\{0, 1\}$, and so on. However, there are many partitions that satisfy the third condition, for example, $P_2 = \{\{0, 1, 2\}, \{3, 4, 5, 6, 7, 8\}, \{9, 10, 11\}, \{12, 13, 14\}\}$. 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 maximum size of a partition subset is bounded by a constant k. By setting $k = 3$, the partition subset $\{3, 4, 5\}$ is preferable to $\{3, 4, 5, 6, 7, 8\}$. The fifth condition says choosing the coarsest partition that is consistent with the first, the second, the third, and the fourth conditions.

Algorithm PAR is shown in Fig. 3. A trace of the algorithm is shown in Table 2. The outer loop processes row 0 to $\ell - 1$. In the first step, the columns of the sorted values in row i are stored in array $R[\ ]$. For $i = 0$, $R[\ ] = \{2, 1, 8, 6, 12, 5, 4, 7, 3, 10, 13, 11, 9, 14, 0\}$. Next, the inner loop tries a number of partition subsets by enlarging $A (A \leftarrow A \cup \{R[j]\})$. If A satisfies condition 3.1, A will be saved

Table 2 A trace of the PAR algorithm

i	j	A	Condition 3.1	B	P
0	0	{0, 2}	True	{0, 2}	Ø
0	1	{0, 2, 1}	True	{0, 2, 1}	{{0,1,2}}
3	0	{3, 4}	False	{3}	{{0, 1, 2}}
3	1	{3, 4, 5}	True	{3, 4, 5}	{{0, 1, 2}, {3, 4, 5}}
6	0	{6, 8}	True	{6, 8}	{{0, 1, 2}, {3, 4, 5}}
6	1	{6, 8, 7}	True	{6, 8, 7}	{{0, 1, 2}, {3, 4, 5}, {6, 7, 8}}
9	0	{9, 11}	True	{9, 11}	{{0, 1, 2}, {3, 4, 5}, {6, 7, 8}}
9	1	{9, 11, 10}	True	{9, 11, 10}	{{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {9, 10, 11}}
12	0	{12, 14}	True	{12, 14}	{{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {9, 10, 11}}
12	1	{12, 14, 13}	True	{12, 14, 13}	{{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {9, 10, 11}, {12, 13, 14}}

The PAR input is the matrix in Fig. 4. The partition subset A is enlarged by $R[j]$, but the size of a partition subset is not allowed to be greater than $k = 3$. If A satisfies condition 3.1, A will be saved to B. Finally, the partition P becomes {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {9, 10, 11}, {12, 13, 14}}

	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8	Col 9	Col10	Col11	Col12	Col13	Col14
Row 0	0	70220	70451	61129	61841	62405	63493	61560	63968	60455	61065	60472	62699	60534	60272
Row 1	70220	0	70130	61115	62569	61972	63075	62080	61943	61290	60002	61259	63515	60205	61223
Row 2	70451	70130	0	62233	63643	62571	64586	64432	64146	61489	61774	61260	63214	61133	62010
Row 3	61129	61115	62233	0	70999	70172	68228	68722	68782	61817	62222	62241	63219	62016	61715
Row 4	61841	62569	63643	70999	0	71543	68738	68474	68064	63443	63244	62739	65128	62765	62995
Row 5	62405	61972	62571	70172	71543	0	68715	68567	68727	62289	62683	62613	63685	62914	62791
Row 6	63493	63075	64586	68228	68738	68715	0	72764	73739	63571	63877	63976	65485	63230	62969
Row 7	61560	62080	64432	68722	68474	68567	72764	0	73045	63215	62996	63359	64957	62862	62538
Row 8	63968	61943	64146	68782	68064	68727	73739	73045	0	63289	63623	63590	66003	63272	63170
Row 9	60455	61290	61489	61817	63443	62289	63571	63215	63289	0	70259	70527	62390	62794	62619
Row 10	61065	60002	61774	62222	63244	62683	63877	62996	63623	70259	0	70457	61318	63258	61094
Row 11	60472	61259	61260	62241	62739	62613	63976	63359	63590	70527	70457	0	63025	61219	63465
Row 12	62699	63515	63214	63219	65128	63685	65485	64957	66003	62390	61318	63025	0	70316	71092
Row 13	60534	60205	61133	62016	62765	62914	63230	62862	63272	62794	63258	61219	70316	0	70832
Row 14	60272	61223	62010	61715	62995	62791	62969	62538	63170	62619	61094	63465	71092	70832	0

Fig. 4 The simultaneity matrix is created by executing the SMC algorithm on a highly fit population. The population is randomly composed of aligned chunks of "000" and "111." The perturbation is omitted because the elements in the upper triangle are distinct. The *elements in the diagonal* are always zero. The matrix is symmetric

to B. Finally, P is the partition that satisfies the five conditions.

Checking condition 3.1 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 $O(\ell^4)$.

4 Experimental results

4.1 Methodology

Most papers report the performance in terms of function evaluations required to reach the optimum. Such a performance measurement is affected by selection method, solution recombination, and the other factors. At present, research community does not provide a formal framework for measuring the effectiveness of a BB identification algorithm regardless of the other factors we have mentioned. Inevitably, we have to make a comparison in terms of function evaluations. We have presented the building-block identification by simultaneity matrix (BISM). An optimization algorithm that exploits the BISM is needed. We customize simple GAs as follows. Every generation, the simultaneity matrix is constructed. The PAR algorithm is executed to find a 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 be passed together. The mutation is turned off. The diversity is maintained by the rank-space method [34, pp. 520–523]. The population size is determined empirically by the bisection method [26, 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 ten runs.

4.2 A visualization of the simultaneity matrix

To illustrate how the matrix changes over time, a matrix element is represented by a square. The square intensity is proportional to the value of matrix element (see Fig. 5). In the early generation (A), the matrix elements are nearly identical because the initial population is generated at random. After that (B), the matrix elements become more distinct. The solution recombination is more speculative. Multiple bits are passed together, and therefore forming larger BBs. Finally (C), the BBs are completely detected. The mixed trap function is additively composed of five-bit onemax, three-, four-, five-, six-, and seven-trap functions. Note that the bits

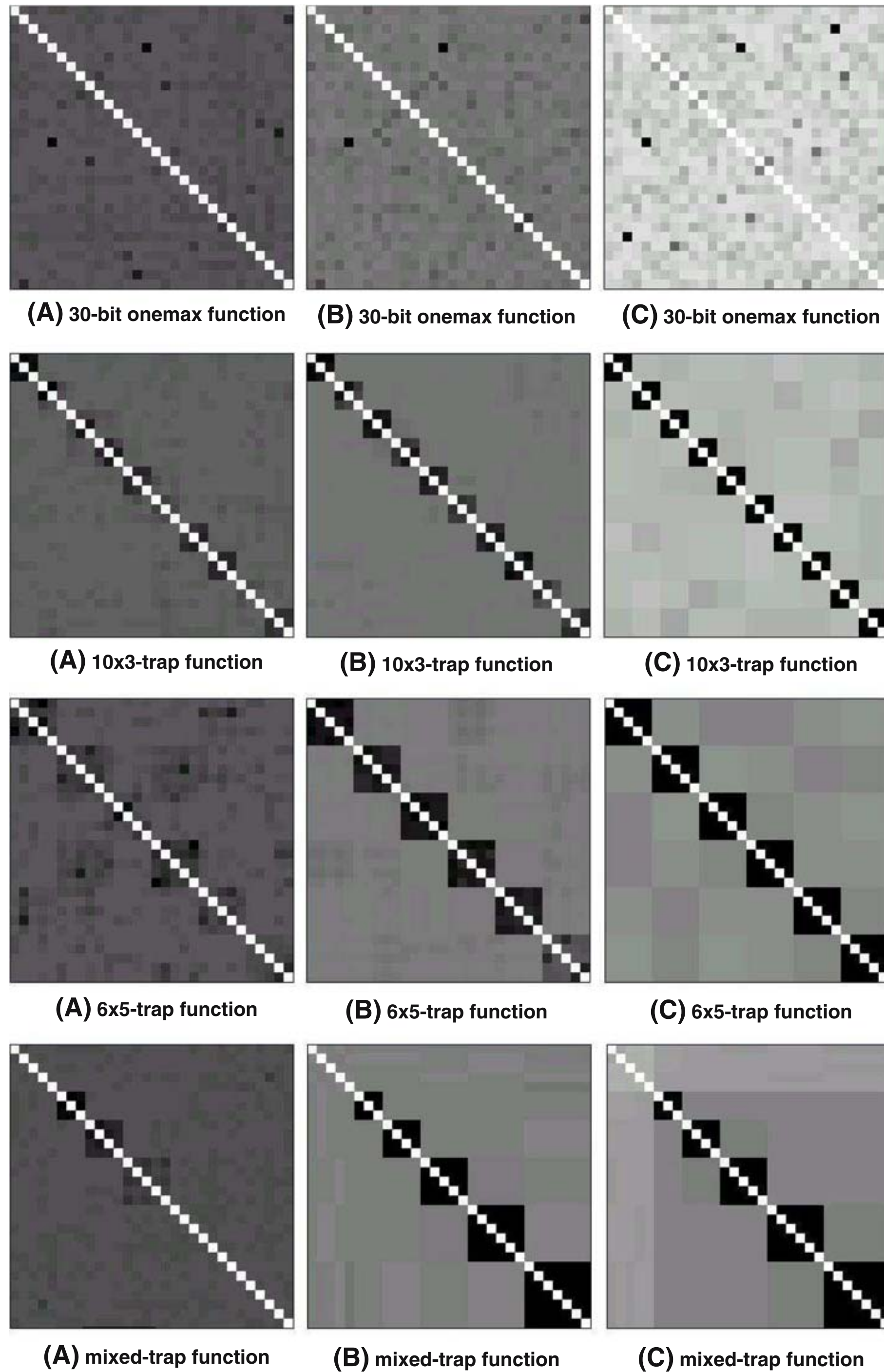

Fig. 5 The simultaneity matrix changes as the population is evolving (onemax, $m \times 3$-trap, $m \times 5$-trap, and mixed-trap functions). Three snapshots are taken for each function (A, B, C)

governed by the same BBs do not need to be packed close together. It is done for the ease of presentation.

4.3 A comparison to the BOA

Our algorithm is compared to the BOA [26, pp. 115–117]. Figures 6, 7, and 8 show the number of function evaluations required to reach the optimal solution. 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 parameter k is known beforehand for the BOA and BISM. The maximum number of incoming edges, a parameter of the BOA [22], limits the number of

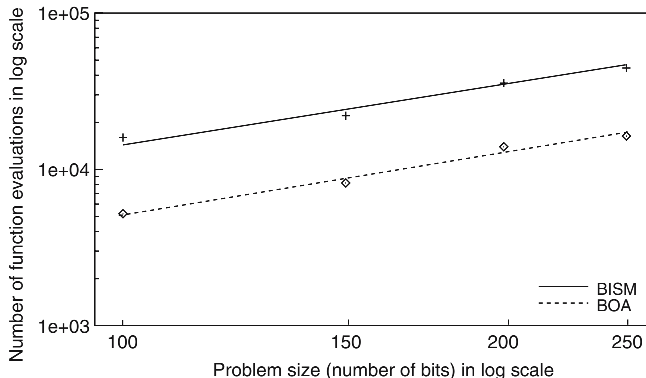

Fig. 6 Performance comparison between the BOA and BISM (onemax, $k = 1$).

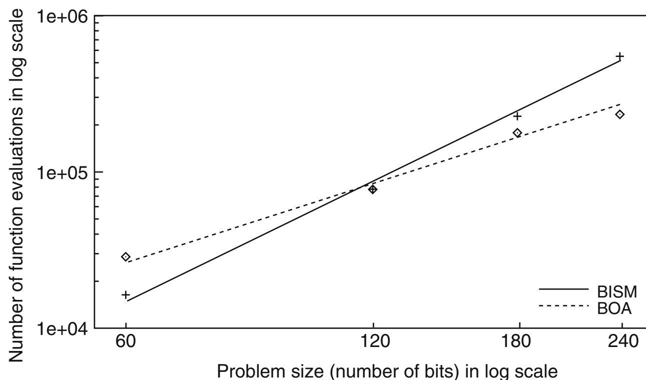

Fig. 7 Performance comparison between the BOA and BISM ($m \times 3$-trap, $k = 3$)

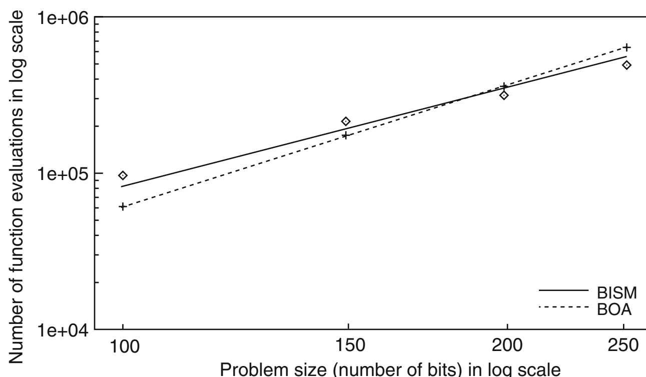

Fig. 8 Performance comparison between the BOA and BISM ($m \times 5$-trap, $k = 5$)

incoming edges for every vertice 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 BISM can solve the ADFs in a polynomial time. The BOA performs better than the BISM. However, the performance gap narrows as the problem becomes harder (onemax, $m \times 3$-trap, and $m \times 5$-trap functions respectively).

We make another comparison in terms of elapsed time. The elapsed time is an execution time of a call on subroutine `constructTheNetwork` [22]. The hardware plat-

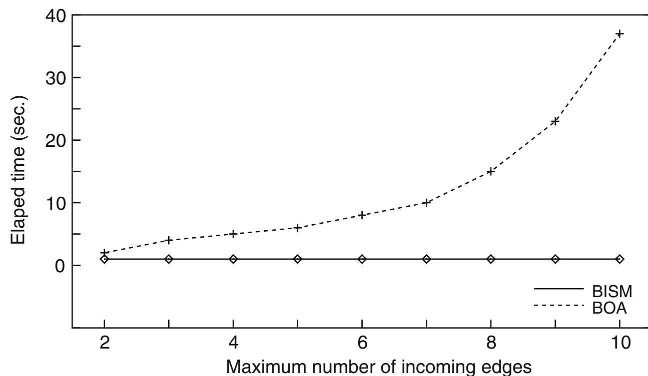

Fig. 9 Elapsed time required to construct Bayesian network (in the BOA) and the upper triangle of the matrix (a half of the matrix is needed because it is symmetric). The problem size and population size are fixed at 250 and 1,200

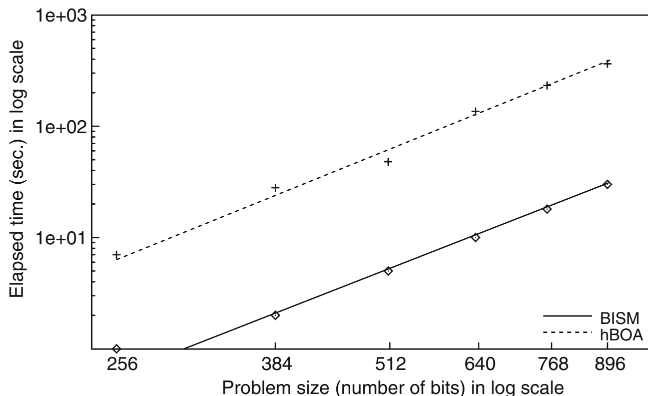

Fig. 10 Elapsed time required to construct Bayesian network (in the hBOA) and the upper triangle of the matrix (a half of the matrix is needed because it is symmetric). The population size is set at three times greater than the problem size

form is HP NetServer E800, 1 GHz Pentium-III, 2GB RAM, and RedHat 8.0 OS. The parameters of the BOA are set at default. Figure 9 shows that the elapsed time required to construct the network increases with the maximum number of incoming edges, but the computational time of the matrix is fixed for a problem size. The difficulty of predetermining the maximum number of incoming edges is resolved in a later version of the BOA, called the hierarchical BOA (hBOA) [25,26]. However, Fig. 10 shows that the hBOA is still time-consuming. This is because the network gathers all statistical dependency between bit variables. In contrast, the matrix records only dependency between two bits that are likely to be disrupted in the uniform crossover. Therefore the matrix computation is much faster.

5 Conclusions

The BB identification is indispensable to the scalability of GAs. We have presented a BB identification by simultaneity matrix. The matrix element m_{ij} is proportional to the

probability that two-bit BBs at positions i and j will be disrupted by the uniform crossover. The matrix does not detect all dependency between bit variables. We have shown that there might be dependency between bits at positions i and j that cannot be detected by the matrix. Such dependency is not necessary because the two-bit BBs at positions i and j are very likely to survive in the next generation regardless of the solution recombination methods. Exploiting the matrix is simply passing the bits at positions i and j together if m_{ij} is significantly high. More formally, we search for a partition of bit positions. The bits governed by the same partition subset are passed together every time performing crossover. It can be shown that the BISM can solve the ADFs in a polynomial relationship between the number of function evaluations and the problem size. More importantly, the matrix computation is simple and fast. Future work is to attack a more difficult problem, called HDFs [3,32].

References

1. Ackley DH (1987) A connectionist machine for genetic hillclimbing. Kluwer, Boston
2. Aporntewan C, Chongstitvatana P (2003) Building-block identification by simultaneity matrix. In: Cant-úPaz E et al (eds) Proceedings of genetic and evolutionary computation conference. Springer, Berlin Heidelberg New York, pp 1566–1567
3. Aporntewan C, Chongstitvatana P (2004) Simultaneity matrix for solving hierarchically decomposable functions. In: Deb K et al (eds) Proceedings of genetic and evolutionary computation conference. Springer, Berlin Heidelberg New York, pp 877–888
4. Baluja S (1994) Population-based incremental learning: a method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, Carnegie Mellon University, Pittsburgh, PA
5. De Bonet JS, Isbell CL, Viola P (1997) MIMIC: finding optima by estimating probability densities. In: Mozer MC, Jordan MI, Petsche T (eds) Advances in neural information processing systems, vol 9. MIT, Cambridge, pp 424–431
6. De Jong KA, Potter MA, Spears WM (1997) Using problem generators to explore the effects of epistasis. In: Bäck T (ed) Proceedings of the 7th international conference on genetic algorithms. Morgan Kaufmann, San Mateo, pp 338–345
7. Goldberg DE (1989) Genetic algorithms in search optimization and machine learning. Addison Wesley, Reading
8. Goldberg DE, Korb B, Deb K (1989) Messy genetic algorithms: motivation, analysis and first results. In: Wolfram S (ed) Complex systems, vol 3, no 5. Complex Systems Publications, Inc., Champaign, pp 493–530
9. Goldberg DE (2002) The design of innovation: lessons from and for competent genetic algorithms. Kluwer, Boston
10. Harik GR (1997) Learning linkage. In: Belew RK, Vose MD (eds) Foundation of genetic algorithms 4. Morgan Kaufmann, San Francisco, pp 247–262
11. Harik GR (1999) Linkage learning via probabilistic modeling in the ECGA. Technical Report 99010, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Champaign, IL
12. Heckerman D, Geiger D, Chickering M (1999) Test function generators as embedded landscapes. In: Banzhaf W, Reeves C (eds) Foundation of genetic algorithms 5. Morgan Kaufmann, San Francisco, pp 183–198
13. Holland JH (1975) Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor, MI
14. Holland JH (2000) Building blocks, cohort genetic algorithms, and hyperplane-defined functions. In: Whitley D (ed) Evolutionary computation, vol 8, no 4. MIT, Cambridge, pp 373–391
15. Kargupta H (1996) The gene expression messy genetic algorithm. In: Proceedings of the IEEE international conference on evolutionary computation. IEEE Press, Piscataway, pp 814–819
16. Kargupta H, Buescher K (1995) The gene expression messy genetic algorithm for financial applications. In: Proceedings of the IEEE/IAFE conference on computational intelligence for financial engineering. IEEE Press, Piscataway, pp 155–161
17. Kargupta H et al (1998) Scalable data mining from distributed, heterogeneous data, using collective learning and gene expression based genetic algorithms. WSU Technical Report EECS-98-001, School of EECS, Washington State University, Pullman, WA
18. Kargupta H, Park B (2001) Gene expression and fast construction of distributed evolutionary representation. In: Whitley D (ed) Evolutionary computation, vol 9, no 1. MIT, Cambridge, pp 43–69
19. Munetomo M, Goldberg DE (1999). Linkage identification by non-monotonicity detection for overlapping functions. In: Whitley D (ed) Evolutionary computation, vol 7, no 4. MIT, Cambridge, pp 377–398
20. Mühlenbein H, Mahnig T (1999) FDA – A scalable evolutionary algorithm for the optimization of additively decomposable functions. In: Whitley D (ed) Evolutionary computation, vol 7, no 4. MIT, Cambridge, pp 353–376
21. Paredis J (1995) The symbiotic evolution of solutions and their representations. In: Eshelman LJ (ed) Proceedings of the 6th international conference on genetic algorithms. Morgan Kaufmann, San Mateo, pp 359–365
22. Pelikan M (1999) A simple implementation of the Bayesian optimization algorithm (BOA) in C++ (version 1.0). Technical Report 99011, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Champaign, IL
23. Pelikan M, Goldberg DE, Cantú-Paz E (1999) BOA: The Bayesian optimization algorithm. In: Banzhaf W et al (eds) Proceedings of genetic and evolutionary computation conference. vol 1. Morgan Kaufmann, San Francisco, pp 525–532
24. Pelikan M, Goldberg DE, Lobo F (1999) A survey of optimization by building and using probabilistic models. In: Hager WW (ed) Computational optimization and applications, vol 21, no 1. Kluwer, Dordrecht, pp 5–20
25. Pelikan M (2000) A C++ implementation of the Bayesian optimization algorithm (BOA) with decision graph. Technical Report 2000025, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Champaign, IL
26. Pelikan M (2002) Bayesian optimization algorithm: from single level to hierarchy. Doctoral dissertation, University of Illinois at Urbana-Champaign, Champaign, IL
27. Pelikan M, Goldberg DE (2003). Hierarchical BOA solves ising spin glasses and MAXSAT. In: Cant-úPaz E et al (eds) Proceedings of genetic and evolutionary computation conference. Springer, Berlin Heidelberg New York, pp 1271–1282
28. Salman AA, Mehrotra K, Mohan CK (2000) Adaptive linkage crossover. In: Whitley D (ed) Evolutionary computation, vol 8, no 3. MIT, Cambridge, pp 341–370
29. Sastry K, Xiao G (2001) Cluster optimization using extended compact genetic algorithm. Technical Report 2001016, Illinois Genetic algorithms Laboratory, University of Illinois at Urbana-Champaign, Champaign, IL
30. Smith J, Fogarty T (1996) Recombination strategy adaptation via evolution of gene linkage. In: Proceedings of the IEEE international conference on evolutionary computation. IEEE Press, Piscataway, pp 826–831
31. Thierens D (1999) Scalability problems of simple genetic algorithms. In: Whitley D (ed) Evolutionary computation, vol 7, no 4. MIT, Cambridge, pp 331–352
32. Watson RA, Pollack JB (1999). Hierarchically consistent test problems for genetic algorithms. In: Angeline PJ, Michalewicz Z, Schoenauer M, Yao X, Zalzala, A (eds) Proceedings of congress on evolutionary computation. IEEE, Piscataway, pp 1406–1413
33. Whitley D, Rana S, Dzubera J, Mathias KE (1996). Evaluating evolutionary algorithms. In Perrault CR, Sandewall E (eds) Artificial intelligence, vol 85, no 1–2. Elsevier, Amsterdam, pp 245–276
34. Winston PH (1992) Artificial intelligence, 3rd edn. Addison-Wesley, Reading