A Note on Building-block Identification by Simultaneity Matrix

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. We propose an approach for identifying BBs. The approach is based on an observation of the compact GA. We observe that the vector elements that correspond to the bits in the same BB are simultaneously updated with a high probability. We formulate the observation to the simultaneity matrix – an $\ell \times \ell$ matrix of numbers constructed from a set of ℓ-bit solutions. The matrix element in row i and column j is denoted by m_{ij}. 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.

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 [7, 4]. 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 splits in 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 Figure 1). It is clear that the solution recombination have to be done, while maintaining the BBs. As a result, the BBs need to be identified explicitly.

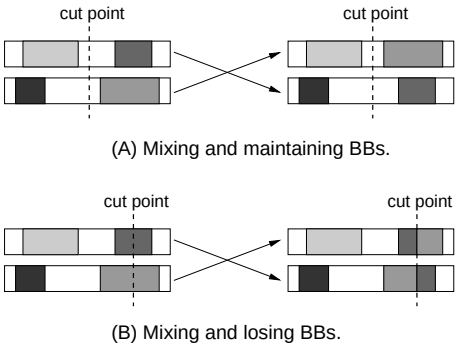

Fig. 1. The solutions are reproduced by the crossover operator. The BBs are shadowed. The cut point, chosen at random, divides a solution into two pieces. Then, the pieces of solutions are exchanged. In case (A), the solutions are mixed while maintaining the BBs. In case (B), the BBs are disrupted.

The trap function [1] is an adversary function for studying BBs and linkage problems in GAs [5]. The general k-bit trap functions are defined as:

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

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:

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

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. 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.

2 An Observation of the Compact GA

The pseudocode of the compact GA is presented in Figure 2 [6]. The compact GA's parameters are population size (n) and solution length (ℓ). A population is represented by an ℓ-dimensional probability vector ($\boldsymbol{p}$). The $p[i]$, that is the i^{th}-element of the probability vector $\boldsymbol{p}$, is the probability that the i^{th} bit of an individual, randomly chosen from the population, will be one. First, $\boldsymbol{p}$ is initialized to $(0.5, \dots, 0.5)$. Next, the individuals a and b are generated according

to $\boldsymbol{p}$. The fitness values, f_a and f_b, are then assigned to a and b respectively. If $f_a \geq f_b$ then the probability vector will be updated towards the individual a. If $a[i] = 1$ and $b[i] = 0$ then $p[i]$ will be increased by $1/n$. If $a[i] = 0$ and $b[i] = 1$ then $p[i]$ will be decreased by $1/n$. The loop is repeated until each $p[i]$ becomes zero or one. Finally, $\boldsymbol{p}$ presents the final solution.

```
1. for i = 0 to ℓ − 1 do p[i] ← 0.5;
2. repeat
       for i = 0 to ℓ − 1 do
           a[i] ← { 1; with probability p[i]
                  { 0; otherwise
           b[i] ← { 1; with probability p[i]
                  { 0; otherwise
       endfor
       f_a ← fitness(a);
       f_b ← fitness(b);
       for i = 0 to ℓ − 1 do
           if f_a ≥ f_b then
               if a[i] = 1 and b[i] = 0 then p[i] ← min(1, p[i] + 1/n);
               if a[i] = 0 and b[i] = 1 then p[i] ← max(0, p[i] − 1/n);
           else
               if a[i] = 1 and b[i] = 0 then p[i] ← max(0, p[i] − 1/n);
               if a[i] = 0 and b[i] = 1 then p[i] ← min(1, p[i] + 1/n);
           endif
       endfor
   until each p[i] ∈ {0, 1}
```

Fig. 2. Pseudocode of the compact GA.

Let $\boldsymbol{p} = (p_0, \ldots, p_{\ell-1})$ be an ℓ-dimensional probability vector where $p_i \in [0, 1]$. Let a simultaneity matrix be an $\ell \times \ell$ matrix of numbers. Let m_{ij} be the matrix element in row i and column j, $0 \leq i, j \leq \ell - 1$. Then m_{ij} is the number of times that p_i and p_j are simultaneously updated. The matrix elements in the diagonal are filled with zeroes. In fact, only half of the matrix is required since it is symmetric. The observation of the compact GA is performed by the following steps.

for $i = 1$ **to** 1000 **do**

1. Randomize a probability vector $\boldsymbol{p}$.
2. Execute Step 2 of the compact GA. After each iterations, record which vector elements are simultaneously updated, in the matrix.

The change of the simultaneity matrix is shown in Figure 3. The fitness function is the 5×3-trap function ($F_{5\times3}$). The population size is set at 100. A matrix element is represented by a square. The square intensity is proportional to the value

of matrix element. In the first iteration (A), the matrix elements are initialized to zeroes. After that (B), the matrix element m_{ij} is increased by one every time p_i and p_j $(i \neq j)$ are simultaneously updated. (C), the matrix elements become more distinct. Finally (D), the matrix is in steady state. An observation is that the vector elements governed by the same 3-bit trap function are simultaneously updated with a high probability.

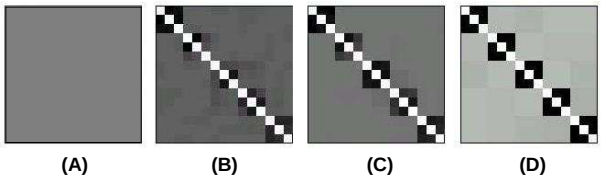

Fig. 3. The simultaneity matrix changes as the compact GA is repeated. The fitness function is the 5×3-trap function ($F_{5\times3}$). Four snapshots are taken (A, B, C, D).

Every iteration of the compact GA, individual a competes with b. If $a[i] \neq b[i]$ and $a[j] \neq b[j]$, the matrix element m_{ij} will be incremented by one. All possible values of $a[i], a[j], b[i], b[j]$ that cause the increment are listed as follows.

$a[i]a[j] = 00$ $b[i]b[j] = 11$, $a[i]a[j] = 11$ $b[i]b[j] = 00$,
$a[i]a[j] = 01$ $b[i]b[j] = 10$, $a[i]a[j] = 10$ $b[i]b[j] = 01$.

The fitness of individuals a and b are improved every iteration. In the later iterations, the fitness of a and b is higher. Roughly speaking, the simultaneity matrix records any pair of 2-bits that are complementary to each other between two highly-fit individuals drawn at random. The highly-fit individuals of the m×3-trap functions are composed of triple zeroes and its complement, triple ones. Thus, we observe the simultaneous change between $p[i]$ and $p[j]$ if bit i and bit j are governed by the same 3-bit trap function. All cases for mixing 2-bit BBs are enumerated. Mixing 00 with 11 results in 01 and 10. Mixing 01 with 10 results in 00 and 11. Only mixing in these cases must be done carefully because the BBs will be lost. Mixing BBs in the other cases gives the same BBs. As a result, it is reasonable to reward a pair of 2-bits that are complementary to each other.

3 The Simultaneity Matrix

In the previous section, it is shown that the simultaneity matrix can be constructed by repeating the compact GA. An alternative is to divide the GAs in two phases. The first phase is to construct the matrix. The second phase performs the optimization by using the BB information in the matrix. A preliminary study shows that computing the matrix by repeating the compact GA consumes a great number of function evaluations. We turn to another way. The BB identification

should not be done separately in the first phase, but it should be co-operated with the optimization phase. For example, the high-quality solutions are shown in Table 1. The fitness is the 5×3-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 high-quality solutions are composed of triple zeroes and triple ones. It is said that the triple zeroes and triple ones are common structures or BBs. The BBs are mixed, hoping to improve the solution quality. Identifying BBs and mixing are repeated until reaching the optimum, and therefore the BBs are inferred from a population every iteration. As mentioned in the previous section, the simultaneity matrix records any pair of 2-bits that are complementary to each other between two high-quality solutions drawn at random. In fact, the matrix can be constructed from a set of solutions. 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, $0 \leq i \leq n-1$. Let $s_i[j]$ be the j^{th} bit of s_i, $0 \leq j \leq \ell - 1$. The matrix is computed as follows.

1. **for** $i = 0$ **to** $\ell - 1$ **do**
   for $j = 0$ **to** $\ell - 1$ **do**
   $m_{ij} \leftarrow 0$;
2. **for** $p = 0$ **to** $n - 1$ **do**
   for $q = p + 1$ **to** $n - 1$ **do**
   for $i = 0$ **to** $\ell - 1$ **do**
   for $j = i + 1$ **to** $\ell - 1$ **do**
   if $s_p[i]s_p[j]$ is complementary to $s_q[i]s_q[j]$ **then**
   $m_{ij} \leftarrow m_{ij} + 1$;
   $m_{ji} \leftarrow m_{ji} + 1$;

The matrix can be computed in $O(\ell^2 n^2)$. A closed form of m_{ij} is defined as:

$$m_{ij} = \mathrm{Count}_S^{00}(i,j)\mathrm{Count}_S^{11}(i,j) + \mathrm{Count}_S^{01}(i,j)\mathrm{Count}_S^{10}(i,j) \tag{3}$$

where $\mathrm{Count}_S^{ab}(i,j) = |\{x \in \{0,\ldots,n-1\} : s_x[i] = a \text{ and } s_x[j] = b\}|$ for all $0 \leq i \leq \ell - 1,\ 0 \leq j \leq \ell - 1,\ (a,b) \in \{0,1\}^2$. By using the closed form, the time complexity of the matrix computation can be reduced to $O(\ell^2 n)$.

Table 1. A set of high-quality solutions is shown in the table. The fitness function is the 5×3-trap function. "111" is the optimum for 3-bit trap function. "000" gives more contribution to the fitness than that of "001," "010," "011," "100," "101," and "110." As a result, the high-quality solutions are composed of "000" and "111."

Sol. no.	$b_0b_1b_2$	$b_3b_4b_5$	$b_6b_7b_8$	$b_9b_{10}b_{11}$	$b_{12}b_{13}b_{14}$
1	111	111	000	111	000
2	000	000	111	000	111
3	111	000	000	111	000
4	000	000	000	000	111
5	000	000	000	000	000

The trap functions 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 a chunk is stored in the matrix. The matrix is not limited to the cases where the two aligned chunks are complementary 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.

4 A Validation of the Simultaneity Matrix

The BB hypothesis states that the solution quality can be improved by composing BBs. The artificial functions are designed so that the BB hypothesis 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 [9, 8].

To compute the HIFF functions, a solution is interpreted as a binary tree. An example is shown in Figure 4. 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:

$$c_i = \begin{cases} 2^h & ; \text{ if node } i \text{ is "0" or "1"} \\ 0 & ; \text{ if node } i \text{ is "-,"} \end{cases} \tag{4}$$

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. 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

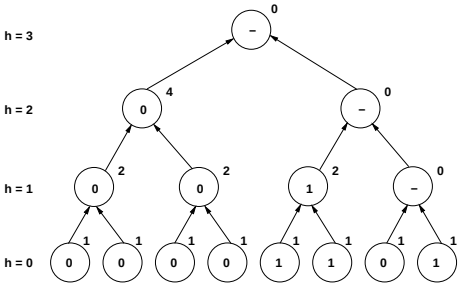

Fig. 4. The HIFF function interprets the solution as a binary tree. The 8-bit solution, "00001101," is placed at the lowest level ($h = 0$). The interpretation results are "0," "1," and "–" according to a deterministic rule. Each node excepting the nodes that are "–"contributes to the fitness by 2^h. The fitness is a total of 18.

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:

$$c_i = \begin{cases} 3^h \cdot F_3(b_0 b_1 b_2) & ; \text{ if } b_j \neq \text{“–”} \text{ for all } 0 \leq j \leq 2 \\ 0 & ; \text{ otherwise,} \end{cases} \tag{5}$$

where h is the height of node i. b_0, b_1, b_2 are the interpretations in the left, middle, right children of node i. At the root node, the trap function's parameters are $f'_{\text{high}} = 1$ and $f'_{\text{low}} = 0.9$. The other nodes use $f_{\text{high}} = 1$ and $f_{\text{low}} = 1$. In Figure 5, the HTrap1 function returns $\sum c_i = 13.05$. The optimal solution is composed of all ones.

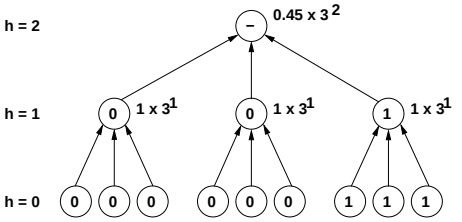

Fig. 5. The HTrap1 function interprets the solution as a 3-branch tree. The 9-bit solution, "000000111," is placed at the lowest level ($h = 0$). The interpretation results are "0," "1," and "–" according to a deterministic rule. Each node excepting the leaf nodes contributes to the fitness by $3^h \cdot F_3(b_0 b_1 b_2)$ where b_i is the interpretation of the child nodes. The fitness of "000000111" is 13.05.

The HTrap2 functions are similar to the HTrap1 functions. The only difference is the trap function's parameters. In the root node, $f'_{\text{high}} = 1$ and

$f'_{\text{low}} = 0.9$. The other nodes use $f_{\text{high}} = 1$ and $f_{\text{low}} = 1 + \frac{0.1}{h}$ where h is tree height. The optimal solution is composed of all ones if the following condition is true.

$$f'_{\text{high}} - f'_{\text{low}} > (h-1)(f_{\text{low}} - f_{\text{high}}) \tag{6}$$

The parameter setting ($f'_{\text{high}} = 1, f'_{\text{low}} = 0.9, f_{\text{high}} = 1, f_{\text{low}} = 1 + \frac{0.1}{h}$) 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 $f_{\text{low}} > f_{\text{high}}$.

To validate the simultaneity 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. A sequence of the simultaneity matrix is shown in Figure 6-7. The population size is set at 50 for all test functions. The onemax function counts the number of ones. The mixed-trap function is composed of 5-bit onemax, 3-bit, 4-bit, 5-bit, 6-bit, and 7-bit trap functions. 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 clear (D). It is seen that the simultaneity matrix is able to identify BBs. 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).

To exploit the matrix, we partition the bit positions $\{0, \ldots, \ell-1\}$ by putting i and j in the same partition subset if the matrix element m_{ij} is high. For instance, the matrix in Figure 3 (D) gives $\{0,1,2\}$,$\{3,4,5\}$,$\{6,7,8\}$,$\{9,10,11\}$,$\{12,13,14\}$. The matrix is computed every generation. Next, the partition is made in $O(\ell^4)$ where ℓ is the solution length. The solution recombination is performed by a restricted uniform crossover – bits governed by same partition subset being passed together. Empirical results show that the matrix is able to solve the ADFs and HDFs in a scalable manner. The number of function evaluations required to reach the optimum grows in a polynomial relationship with the problem size (for more details, see [3]).

5 Conclusions

We have presented how we discover the simultaneity matrix. The discovery is based on the observation of the compact GA. The matrix element m_{ij} is the degree of dependency between bit i and bit j. The time complexity of computing the matrix is $O(n\ell^2)$ where n is the number of solutions and ℓ is the solution

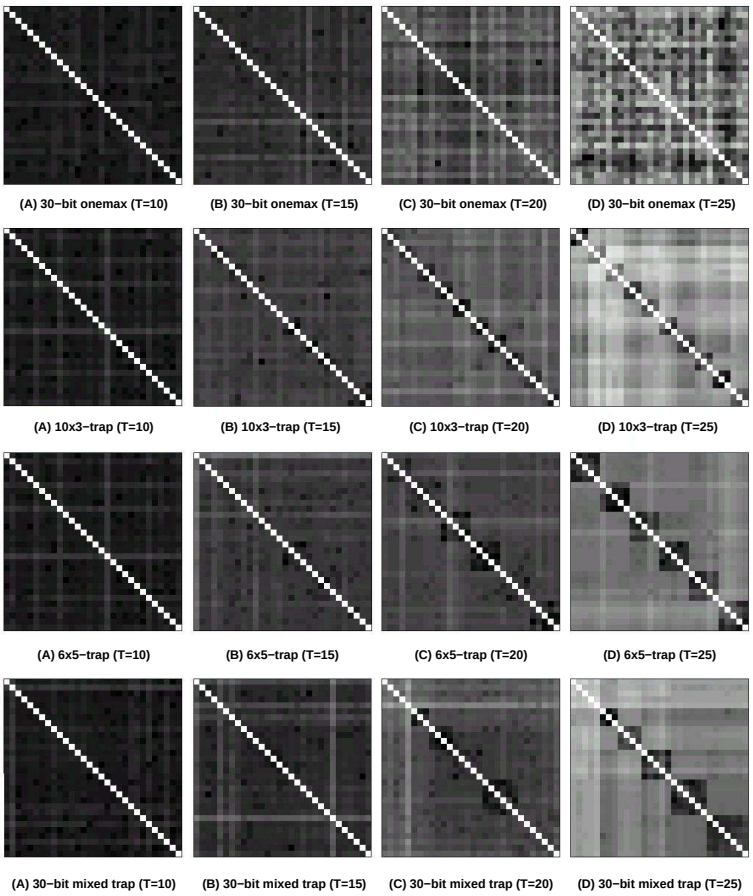

Fig. 6. The simultaneity matrix constructed from a set of 50 random solutions. The fitness of any individual in the population is greater than the threshold T. The ADFs have only single-level BBs.

length. 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 simultaneity 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 (see [3]).

References

1. Ackley, D. H. (1987). A Connectionist Machine for Genetic Hillclimbing. Kluwer Academic Publishers, Boston, MA.

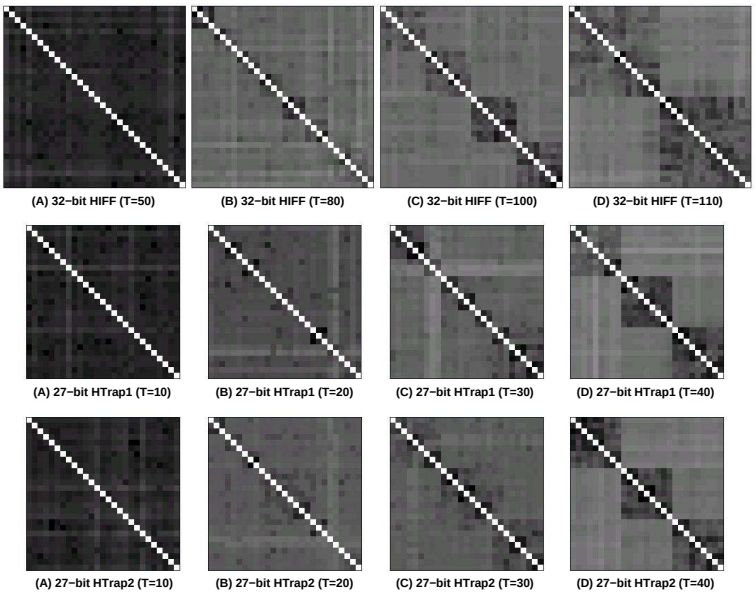

Fig. 7. The simultaneity matrix constructed from a set of 50 random solutions. The fitness of any individual in the population is greater than the threshold T. The HDFs have multiple-level BBs.

2. Aporntewan, C., and Chongstitvatana, P. (2003). Building-block identification by simultaneity matrix. In CantúPaz, E. et al., editors, Proceedings of the Genetic and Evolutionary Computation, page 1566–1567, Springer-Verlag, Heidelberg, Berlin.
3. Aporntewan, C., and Chongstitvatana, P. (2004). Simultaneity matrix for solving hierarchically decomposable functions. In Deb, K. et al., editors, Proceedings of the Genetic and Evolutionary Computation, Springer-Verlag, Heidelberg, Berlin (to appear).
4. Goldberg, D. E. (1989). Genetic Algorithms in Search Optimization and Machine Learning. Addison Wesley, Reading, MA.
5. Harik, G. R. (1997). Learning linkage. In Belew, R. K., and Vose, M. D., editors, Foundation of Genetic Algorithms 4, page 247–262, Morgan Kaufmann, San Francisco, CA.
6. Harik G. R., Lobo F. G., and Goldberg, D. E. (1999). The compact genetic algorithm. In Fogel, D. B., editor, IEEE Transaction on Evolutionary Computation, Vol. 3, No. 4, page 287–297, IEEE Press, Piscataway, NJ.
7. Holland, J. H. (1975). Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, MI.
8. Pelikan, M. (2002). Bayesian optimization algorithm: From single level to hierarchy. Doctoral dissertation, University of Illinois at Urbana-Champaign, Champaign, IL.
9. Watson, R. A., and Pollack, J. B. (1999). Hierarchically consistent test problems for genetic algorithms. In Angeline, P. J., Michalewicz, Z., Schoenauer, M., Yao, X., and Zalzala, A., editors, Proceedings of Congress on Evolutionary Computation, page 1406–1413, IEEE Press, Piscataway, NJ.