Solving Additively Decomposable Functions by Building Blocks Identification

W. Punyaporn, J. Ponsawat, and P. Chongstitvatana
Department of Computer Engineering,
Chulalongkorn University, Bangkok 10330, Thailand
wandao.p@gmail.com, jiradej.p@student.chula.ac.th, and prabhas@chula.ac.th

***Abstract*- This paper proposes a way to use Building Blocks to improve solutions in Genetic Algorithm. Hard problems, for instance, Additively Decomposable Functions (ADFs) cannot be effectively solved by a standard algorithm such as Simple Genetic Algorithm. A single point crossover creates disruption of good solutions for such problems. We proposed using Building Blocks Identification and performed appropriate crossover to solve ADFs. The experiment shows the validity of the proposed method.**

I. Introduction

Building Blocks (BBs) are an important concept in Genetic Algorithm, according to the Schema theorem [5], [6], [9], [10]. BB is composed of two parts of definition. First, BB is embedded in the solution with high fitness. Second, properly composing these BBs gives the solution with higher fitness. However, BBs are not easily identified. This research proposed an approach to identify BBs in form of highly-related-group of bits as partitions. The knowledge of BBs can be used to prevent disruption of highly fit solutions from crossover operators. When performing crossover, group of bits in the same BB should not be divided.

Building Blocks can be identified by computing Chi-square Matrices and use Partition Algorithm proposed in [3]. Each element of Chi-square matrices represents the degree of relation between two bits of selected population. Partition Algorithm groups bits with high relationship into BBs.

To validate our hypothesis, we conduct experiments using problems of Additively Decomposable Functions (ADFs) (see Section III for the definition) which evidently consisted of BBs. The proposed method consisted of identifying BBs and using the knowledge of partitions to perform the appropriate crossover. This method is tested against a standard method, Simple Genetic Algorithm (SGA).

ADFs are hard problems for SGA because a single-point crossover in SGA disrupts good solutions very often. The proposed method conserves good solutions and composes them into better solutions.

Many recent papers are still interested in using various styles of GA to solve ADFs. In [7], GA is modified to use an adaptive population size. Furthermore, it also uses ADFs to compare adaptive GA with SGA. Other papers using trap functions to test the performance of their algorithms such as [11] and [12].

Some work encouraged finding BBs in order to improve GA's functions such as Linkage Learning [8]. Many direct methods to find BBs are proposed such as [2].

We use the method in [2] to find BBs and show how to perform crossover using Building Blocks. Similar idea can be found in [4] where BBs are used to guide updating rules for probabilistic model building GA. However, our work is unique in the sense of applying BB concept to perform crossover directly.

II. Building Block Identification

The algorithm presented in this paper is divided into two parts, the Chi-Square Matrix construction and the Partition algorithm (PAR).

A. Chi-square Matrix

The quantity of building blocks inversely relates to randomness. The Chi-square Matrix [1] is chosen for measuring randomness because computing the matrix is simple and fast.

Let M = (m_{ij}) be an $l \times l$ symmetric matrix of numbers. Let P be a population or a set of l bit binary strings. The Chi-square matrix is defined as follows.

$$\begin{aligned} m_{ij} &= ChiSquare(i,j) && ; \text{if i} \neq \text{j} \\ &= 0 && ; \text{otherwise} \end{aligned} \qquad (2.1)$$

The *ChiSquare(i,j)* is defined as follows.

$$\sum_{xy} \frac{(Count_P^{xy}(i,j) - n/4)^2}{n/4}, \; xy \in \{00, 01, 10, 11\} \qquad (2.2)$$

Where the observe frequency $Count_P^{xy}(i,j)$ counts the number of solutions in which bit i is identical to x and bit j is identical to y. The expected frequencies of observing "00,""01," "10," "11" are $n/4$ where n is the number of solutions. 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 $O(l^2 n)$.

B. Partitioning (PAR) Algorithm

Partitioning (PAR) Algorithm [3] will partition each input bit into suitable blocks. When performing crossover, bits in the same partition must not be separated. The PAR input is an $l \times l$ matrix and its outputs the partition:

$$P = \{B_0, \ldots, B_{|P|-1}\}, \bigcup_{i=0}^{|P|-1} B_i = \{0 \ldots\ldots l-1\},$$
$$B_i \cap B_j = \varnothing \text{ for all } i \neq j. \qquad (2.3)$$

The B_i is called partition subset. There are several definitions of the desired partition. Algorithm PAR must have some preconditions.

1. P is a partition.
   The members of P are disjoint set.
   The union of all members of P is $\{0, \ldots, l-1\}$.
2. $P \neq \{\{0, \ldots, l-1\}\}$.
3. For all $B \in P$ such that $|B| > 1$,
   For all i $\in$ B, the largest $|B| - 1$ matrix elements in row i are founded in columns of B \ {i}.
4. For all $B \in P$ such that $|B| > 1$,
   $H_{max} - H_{min} < \alpha(H_{max} - L_{min})$ where $0 \leq \alpha \leq 1$,
   $H_{max} = max(\{m_{ij} \mid (i, j) \in B \times B, i \neq j\})$,
   $H_{min} = min(\{m_{ij} \mid (i, j) \in B \times B, i \neq j\})$, and
   $L_{min} = min(\{m_{ij} \mid i \in B, j \in \{0, \ldots. l-1\} \setminus B\})$.
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, second, third and fourth conditions, $B \subset B_x$.

All the partition subsets can expand until they satisfy one of the preconditions above.

M= (m_{ij}) denotes $l \times l$ Chi-Square matrix. $0 \leq i, j \leq l-1$.
T_i and $R_{i,j}$ denote arrays of numbers indexed by $0 \leq i, j \leq l-1$.
A and B are partition subsets. P denotes a partition.
Algorithm PAR(M, α)

$P \leftarrow \varnothing$;
for $i = 0$ **to** $l - 1$ **do**
  if $i \notin B$ for all $B \in P$ **then**
    array T = {matrix elements in row i sorted in descending order};
    for $j = 0$ **to** $l - 1$ **do**
      $R_{i,j} = x$ where $m_{ix} = T_j$
    endfor
    $A \leftarrow \{i\}$;
    $B \leftarrow \{i\}$;
    for $j = 0$ **to** $l - 3$ **do**
      $A \leftarrow A \cup \{R_{i,j}\}$;
      if A satisfies the third and the fourth conditions **then**
        $B \leftarrow A$;
      endif
    endfor
    $P \leftarrow P \cup \{B\}$;
  endif
endfor
return P;

C. *Crossover Method*

The crossover operator can exploit the knowledge of BBs by choosing appropriate cut points. The cut point should not separate bits in the same BB (see Fig. 1). To achieve this, a crossover mask in created for each partition. When parents exchange bits to create offspring, all bits in the same partition will be moved together. See the following example:

Partition	<1 2 3 4 1 3 2 4 5 6 7 8 5 4 6>
Mask Bits	<0 1 1 1 0 1 1 1 0 1 1 1 0 1 1>

x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Parent 1

y	y	y	y	y	y	y	y	y	y	y	y	y	y	y

Parent 2

0	1	1	1	0	1	1	1	0	1	1	1	0	1	1

Mask Bits

After crossover, the two parents produce two children.

y	x	x	x	y	x	x	x	y	x	x	x	y	x	x

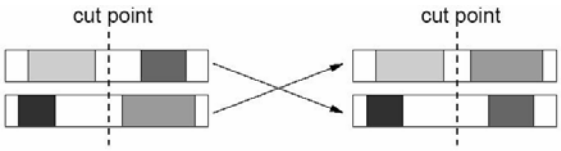
Child 1

x	y	y	y	x	y	y	y	x	y	y	y	x	y	y

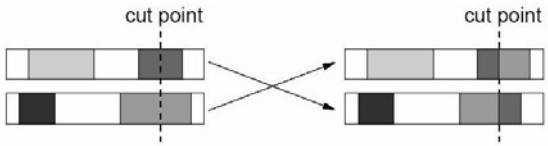
Child 2

The number in the partition shows the relation between bits. The same number illustrates the same partition which is also in the same building block. Flip coin method is used to choose whether partitions will be removed or remain unchanged. For instance, if the partition "1" is assigned to 0, all parts labeled with "1" are also assigned 0. After assign 0/1 to all partitions in each gene, the partitions which are assigned to 0 must be swapped to their mate. Otherwise, they remain in the same positions.

Figure 1 illustrates the difference between the crossover of BB algorithm and the crossover of SGA. The former will not break into a partition while the latter randomly chooses the cut point without considering the building block.



(A) Mixing and maintaining BBs



(B) Mixing and losing BBs

Figure 1 Building block characteristics

III. BENCHMARK PROBLEM

To validate the proposed method, a set of benchmark is tested. The ADFs are chosen because they evidently consisted of BBs. The standard ADF is Trap function. To emphasize the effect of BBs, we also construct a modified version called Shuffle Trap function. The Shuffle Trap function has "loose-coding" of solution, i.e. bits in a BB is positioned far apart. Hence the single point crossover is ineffective against this encoding. It always disrupts the BBs. The definitions of Trap functions are as follows.

m × *k-trap function*

m × *k*-trap function is defined as following.

$$F_{m\times k} : B \rightarrow R,\ B \in B_0 \ldots B_{m-1},\ B_i \in \{0,1\}^k \tag{3.1}$$

$$F_{m\times k}(B_0 \ldots B_{m-1}) = \sum_{i=0}^{m-1} F_k(B_i) \tag{3.2}$$

Where F_k is *k-trap* function [11]. The *m* and *k* are varied to produce many test functions. These functions are often referred to as additively decomposable functions (ADFs). The optimal solution consists of all "1" bits.

Shuffle m × *k-trap function*

The shuffle trap function is constructed by spreading bits of the same building blocks. For instance, normal 4×5-trap function has building blocks as shown.

11111 xxxxx xxxxx xxxxx

The modulo method is used to construct one building block. The bits in the same building block are spreading out every *m* bits.

1xxx 1xxx 1xxx 1xxx 1xxx

The trap functions are composed of:

a) 20×3- trap function
b) 20×3- shuffle bits trap function
c) 10×4- trap function
d) 10×4- shuffle bits trap function

To find solutions to the problems, the parameters for a) and b) are set as follows: population size = 15000, max generation = 500, crossover rate = 0.9, mutation rate is turned off and threshold (α) in PAR is set to 0.95 while the parameters for c) and d) are set as follows: population size = 50000, max generation = 500, crossover rate = 0.9, mutation rate is turned off for BB algorithm while SGA is assigned to 0.3. Threshold required in creating partition subset is set to 0.95.

IV. Experimental Result

For comparison, SGA is used to solve for these problems. Each graph is averaged from 25 independent runs. Figure 2 shows the relation between generations and fitness value in normal 3×20-trap function. The results illustrate that BB crossover algorithm performs better with respect to the mean and maximum fitness value than SGA. When increase one bit to each trap set in figure 3, the difference is even more pronounced. BB algorithm first found the optimal solution in the 60th generation and reached the steady state in the 140th generation. On the other hand, SGA got stuck in this deceptive function and cannot reach the optimal solution.

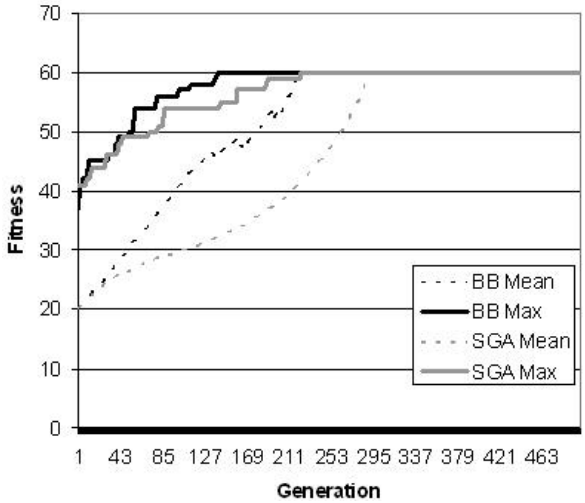


Figure 2 Simulation result for 20×3-trap function

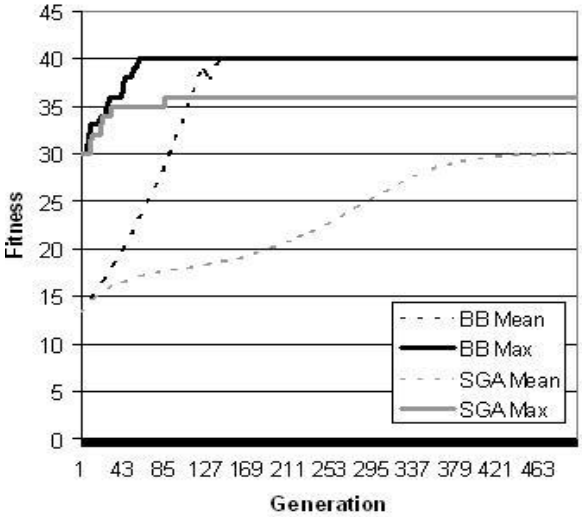


Figure 3 Simulation result for 10×4-trap function

Figure 4 and 5 represent the results of the shuffle trap function. Figure 4 clearly shows that BB gains higher performance than SGA. BB can reach the optimal solution, while SGA cannot. The situation is even worse for SGA with the larger size problem. The result is shown in figure 5.

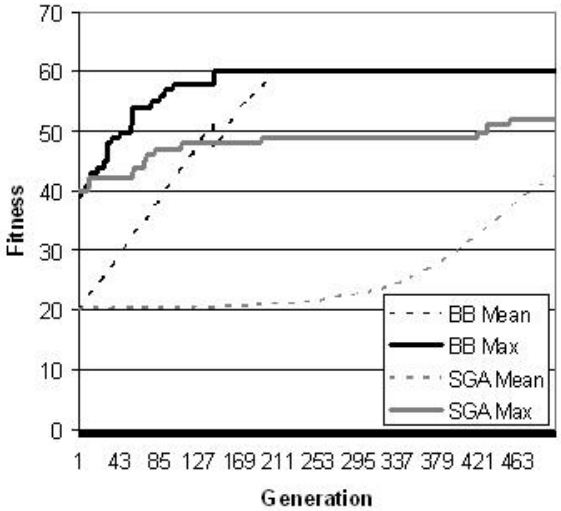


Figure 4 Simulation result for 20×3-shuffle bit trap function

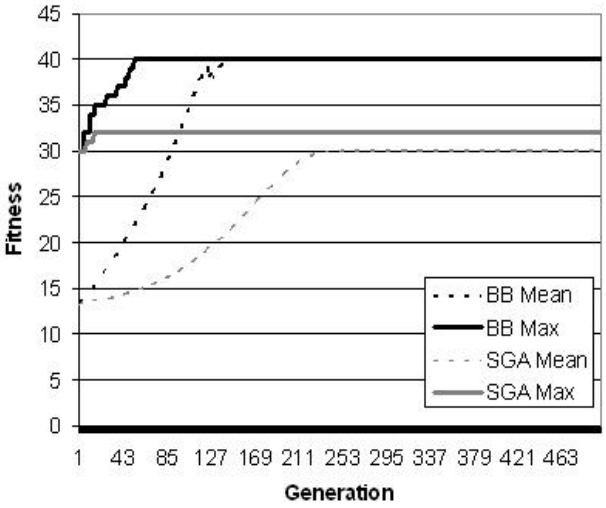


Figure 5 Simulation result for 10×4-shuffle bit trap function

From results of the experiments, the proposed algorithm reaches the optimal solution in earlier generation than SGA in normal trap function. Moreover, in shuffle trap function, SGA cannot find the optimal solution while our algorithm can.

V. CONCLUSION

We show that BB crossover algorithm can solve many deceptive functions, 20×3-bit trap, shuffle 20×3-bit trap, 10×4-bit trap and shuffle 10×4-bit trap. As a result, our algorithms perform more effectively than SGA. Future work will explore the quality of BB algorithms in multiple objective optimization problems as well as in a wide range of real-world applications.

REFERENCES

[1] C. Aporntewan and P. Chongstitvatana, A quantitative approach for validating the building block hypothesis, IEEE Congress of Evolutionary Computation, Edinburgh, September 2-5, 2005.
[2] C. Aporntewan and P. Chongstitvatana, Chi-square matrix: An approach for building-block identification. In Proceedings of 9th Asian Computing Science Conference, December 8-10, 2004, 63–77.
[3] C. Aporntewan and P. Chongstitvatana, Simultaneity matrix for solving hierarchically decomposable functions. Proceedings of the Genetic and Evolutionary Computation, Springer-Verlag, Heidelberg, Berlin,2004, 877–888.
[4] C.F. Lima and K. Sastry, Combining Competent Crossover and Mutation Operators: a Probabilistic Model Building Approach, GECCO'05, Washington, DC, USA, June 25–29, 2005.
[5] D. E. Goldberg, Genetic Algorithms in Search Optimization and Machine Learning. Addison Wesley, Reading, MA, 1989.
[6] D. E. Goldberg, The Design of Innovation: Lessons from and for Competent Genetic Algorithms. Kluwer Academic Publishers, Boston, MA, 2002.
[7] F.G. Lobo and C.F. Lima , Revisiting Evolutionary Algorithms with On-the-Fly Population Size Adjustment, GECCO'06, Seattle, Washington, USA, July 8–12, 2006, 1241-1248.
[8] G. R. Harik, Learning Linkage, Foundation of Genetic Algorithms 4, Morgan Kaufmann, San Francisco, 1997, 247-262.
[9] J. H. Holland, Adaptation In Natural and Artificial Systems, University of Michigan Press, 1975.
[10] J. H. Holland, Building blocks, cohort genetic algorithms, and hyperplane defined functions. Evolutionary Computation, 8(4), MITPress, Cambridge, MA, 2000, 373–391.
[11] M. Clergue and P. Collard, GA-hard functions built by combination of Trap functions. In Proceedings of the 2002 Congress on Evolutionary Computation, May 12-17, 2002.
[12] S. Nijssen and T. Back, An analysis of the behavior of simplified evolutionary algorithms on trap functions. IEEE Transactions on Evolutionary Computation, 7(1), February, 2003.