Adaptive Discretization for Probabilistic Model Building Genetic Algorithms

Chao-Hong Chen
Dept. of Computer Science
National Chiao Tung University
HsinChu City 300, Taiwan
chaohung@csie.nctu.edu.tw

Wei-Nan Liu
Depart. of Computer Science
National Chiao Tung University
HsinChu City 300, Taiwan
wnliu@csie.nctu.edu.tw

Ying-Ping Chen
Depart. of Computer Science
National Chiao Tung University
HsinChu City 300, Taiwan
ypchen@cs.nctu.edu.tw

ABSTRACT

This paper proposes an adaptive discretization method, called *Split-on-Demand* (SoD), to enable the probabilistic model building genetic algorithm (PMBGA) to solve optimization problems in the continuous domain. The procedure, effect, and usage of SoD are described in detail. As an example, the integration of SoD and the extended compact genetic algorithm (ECGA), named *real-coded* ECGA (rECGA), is presented and numerically examined. The experimental results indicate that rECGA works well and SoD is effective. The behavior of SoD is analyzed and discussed, followed by the potential future work for SoD.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning—*Parameter learning*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Algorithms

Keywords

Adaptive discretization, split-on-demand, extended compact genetic algorithm, real-parameter optimization

1. INTRODUCTION

Genetic algorithms (GAs) [6, 2] are methodologies inspired by Darwinian evolution and designed according to the biological genetic operations. As a flexible optimization tool, genetic algorithms are nowadays widely applied to tackle a number of real-world optimization problems. In principle, genetic algorithms select good, promising individuals from the current population and generate new candidate of solutions by employing recombination and mutation.

According to the theory of design decomposition [3], the key components to the GA success include identifying, reproducing, and exchanging the structure of the solutions. Recombination, one of the main GA operator, mixes the promising sub-solutions, called building blocks (BBs), and creates the solutions. Genetic algorithms therefore work very well for the problems which can be somehow decomposed into sub-problems. However, the problem-independent recombination operator with fixed chromosome representations often breaks building blocks and results in ineffective mixing. It is the reason when traditional genetic algorithms meet complex solution structures which consist of a group of related genes, they oftentimes fail to effectively identify and efficiently exchange the building blocks to create good final solutions [4].

In order to appropriately mix genes, the evolutionary algorithms based on utilizing probabilistic models were proposed and developed [7, 9]. In such schemes, the offspring population is generated according to the estimated probabilistic model of the parent population instead of using regular recombination and mutation operators. The probabilistic model is expected to reflect the problem structure, and better performance can be achieved via exploring and exploiting the relationship between genes. These evolutionary algorithms are called probabilistic model building genetic algorithms (PMBGAs) or estimation of distribution algorithms (EDAs) [7, 9].

In PMBGAs, decision variables are often coded with binary coding or gray coding. However, it is reportedly difficult to find high accuracy solution in solving continuous problems. Moreover, many real-world engineering problems are real-parameter optimization problems, such as structural optimization problems and the design of a transonic wing of an aircraft. In the literature, several attempts to apply PMBGAs to problems in the continuous domain have been made, including continuous PBIL with Gaussian distribution [11], real-coded variant of PBIL with interval updating [12], BEA for continuous function optimization [13], and the real-coded BOA [1]. In this paper, we propose a framework that can enable the PMBGAs designed for handling bit-strings to tackle real-valued optimization problems. Particularly, we develop a new, adaptive discretization encoding scheme that can be easily integrated into PMBGAs, and we use the extended compact genetic algorithm (ECGA) [5] as an example in the present work.

In next section, we will first briefly introduce ECGA. In section 3, we will describe in detail how the proposed *Split-on-Demand* (SoD) encodes the solutions of real values into discrete numbers. In section 4, we use SoD to enable ECGA to handle real-valued decision variables and test the inte-

GECCO'06, July 8–12, 2006, Seattle, Washington, USA.

grated framework with the benchmark proposed in the special session on real-parameter optimization in CEC 2005 in section 5. Finally, section 6 concludes this work.

2. BRIEF REVIEW OF ECGA

As a study subject, we will first briefly review the *extended compact genetic algorithm* (ECGA) in this section. ECGA, proposed by Harik [5], is based on the idea that the choice of a good probability distribution is equivalent to learning genetic linkage. The probabilistic models used in ECGA are a class of probabilistic models known as the marginal product models (MPMs). ECGA uses MPMs to model partitions of the decision variables. The measure of good distributions is quantified based on the minimum description length (MDL) principle [10]. The key concept of the MDL model is that all things being equal, simpler distributions are better than more complex ones. The MDL restriction penalizes both inaccurate and complexity, thereby leading to an optimal probability distribution.

The ECGA can be algorithmically outlined as:

1. Generate a random population of size N.
2. Apply the tournament selection at a rate S.
3. Model the population using a greedy MPM search.
4. If the model has converged, stop.
5. Generate a new population using the given model.
6. Return to step 2.

The complexity measure of MPM is the sum of Model Complexity and Compressed Population Complexity. By the MDL principle, we wish to minimize the Combined complexity. Combined Complexity = Model Complexity + Compressed Population Complexity.

$$Model\ Complexity = \log N \sum_I 2^{S[I]} ,$$

where N is the population size, and $S[I]$ is the length of the Ith subset of genes.

$$Compressed\ Population\ Complexity = N \sum E(M_I) ,$$

where $E(M_I)$ is the entropy of the marginal distribution of subset I.

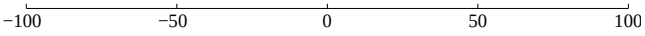
Instead of applying traditional crossover and mutation operators, ECGA generates the new population from the MPM obtained in step 3. In this way, new individuals are generated without breaking building blocks. In the original design of ECGA, the framework can only deal with bit-strings. In next section, we will propose the encoding method and integrate it into ECGA in section 4.

3. SPLIT-ON-DEMAND

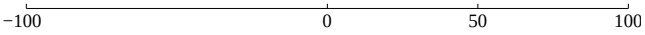
In this section, we present an encoding method called *Split-on-Demand* (SoD), which can encode real-coded decision variables into discrete numerical values. The main idea of Split-on-Demand is to split the interval where we demand to know in more detail and to build a more accurate probabilistic model with the information obtained during the search process. Because of the behavior to split the interval which needs further investigation, we call the proposed encoding scheme *Split-on-Demand*.

Dimension 1				Dimension 2			
Interval			Code	Interval			Code
-100	∼	-50	0	-100	∼	0	0
-50	∼	0	1	0	∼	50	1
0	∼	50	2	50	∼	100	2
50	∼	100	3				

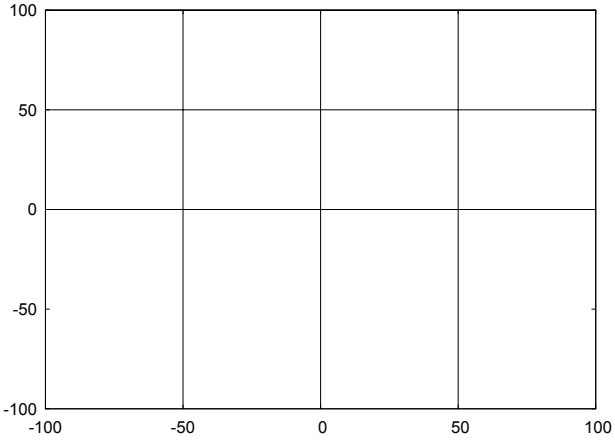
Figure 1: An example code table constructed by Split-on-Demand for a real-parameter optimization problem of two dimensions.



(a) Split configuration on dimension 1.



(b) Split configuration on dimension 2.



(c) Combined split configuration on both dimensions.

Figure 2: Illustration of the solution space split according to the code table given in Figure 1.

As described, SoD splits a dimension of real numbers into several intervals and gives each of them an integer code. We can then translate a vector of real numbers to a vector of integers, which can be represented by bits or binary codes more intuitively. As an example, given a real-parameter optimization problem of two dimensions, one possible code table constructed by SoD is shown in Figure 1. According to the code table, the solution $[-72.3, 24.8]$ is encoded as $[0, 1]$, and the solution $[13.8, -5.3]$ as $[2, 0]$. Figure 2 shows the solution space split by the code table given in Figure 1 as an illustration. Figure 2(a) is the split configuration on dimension 1, Figure 2(b) is the split configuration on dimension 2, and Figure 2(c) is the combined split configuration on [dimension 1, dimension 2], which is the whole solution space. The code table splits the solution space into 12 regions.

After describing the usage of the SoD code table, we now discuss the way to construct it. The principle of the proposed encoding scheme is to split the real number interval in which there are a lot of search points. Because the tournament selection operator is applied to choose the promising individuals at each generation, if there are a number of individuals in certain region after selection, we consider that

```
1: procedure SPLIT-ON-DEMAND
2:     Split(low_bound, high_bound)
3:     γ ← γ × ϵ
4: end procedure

1: procedure SPLIT(low, high)
2:     mid ← random[low, high]
3:     num_low ← number of individuals in [low, mid]
4:     num_high ← number of individuals in [mid, high]
5:     if num_low ≥ N × γ then
6:         Split(low, mid)
7:     else
8:         Add_Code(low, mid)
9:     end if
10:    if num_high ≥ N × γ then
11:        Split(mid, high)
12:    else
13:        Add_Code(mid, high)
14:    end if
15: end procedure
```

Figure 3: Pseudo code of SoD.

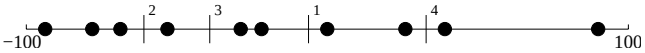
region important and believe the probability to find good solutions in that region is higher. Therefore, we split the promising region to gain higher resolution as well as achieve better accuracy in order to assist the back-end PMBGA to build high quality probabilistic models.

In order to determine which real number interval to split, we employ a *split rate* γ, where $0 < \gamma < 1$. Assume that the population size is N, if an interval contains more than $N \times \gamma$ individuals, the interval should be split. By adjusting the split rate, we can control the accuracy of the probabilistic model which we want to build. If more accurate probabilistic models are necessary, smaller split rates should be used such that the value range of the decision variables is split to more intervals. Furthermore, for the same reason, the split rate can also be used to control the code length. The higher the split rate, the shorter the code length, and vice versa.

The procedure of Split-on-Demand can be describe as follows, and the pseudo code of SoD is shown in Figure 3. Subroutine `Split-on-Demand` first calls subroutine `Split` on the interval $[low\_bound, high\_bound]$, where the low_bound and $high\_bound$ are the bounds of this dimension. `Split` generates a random number mid in the interval in question and counts the individuals in the two intervals: $[low\_bound, mid]$ and $[mid, high\_bound]$. If an interval contains more than $N \times \gamma$ individuals, `Split` will be recursively called to split that interval until no interval should be further split.

When all split operations are done, we decrease the split rate by a factor ϵ, where $0 < \epsilon < 1$. The reason to decrease the split rate is to have a higher split rate to keep the diversity and implement a coarse-grained, global search at the early stage of search. As the search process goes by, we obtain more and more information about the solution space and know where to put more search points to find good solutions. Hence, at the late stage of search, a lower split rate is needed to build accurate probabilistic models for conducting a fine-grained, local search. The factor ϵ can be set to control the speed of convergence. An appropriate ϵ can help the search algorithm to avoid wasting time on useless regions as well as being trapped at local optima and therefore is key to an efficient search process.

(a) Population distribution and 2 split positions at generation 1. $\gamma = 0.5$. $10 \times \gamma = 5$.



(b) Population distribution and 4 split positions at generation 10. $\gamma = 0.4$. $10 \times \gamma = 4$.

(c) Population distribution and 5 split positions at generation 20. $\gamma = 0.3$. $10 \times \gamma = 3$.

Figure 4: Population distribution and the split positions at different generations.

We now give a typical example of how SoD run on the population for demonstration. Assume that the population size is 10, and the initial split rate $\gamma = 0.5$. Figure 4 depicts how the individuals distributed at different generations. Initially, Figure 4(a) shows that the first position to split, marked by 1, is randomly generated. We then discover that the number of individuals in the left interval is larger than $10 \times \gamma = 5$. Under this condition, SoD calls `Split` to perform a random split on the left interval and gets the second split position, marked by 2. After the second split, the numbers of individuals in the two intervals, the left interval and the right interval to the second split position, are both less than $10 \times \gamma = 5$. As a consequence, SoD stops the split operation and decreases the split rate.

Figure 4(b) is the population distribution and the split positions at generation 10. The split rate γ is now 0.4. Similar to the procedure described in the previous paragraph, SoD performs a random split to cut the whole interval into two intervals. It can be observe that both the left and the right intervals contain more than $10 \times 0.4 = 4$ individuals, and as a result, SoD calls `Split` on both the left and the right intervals. For the left interval, SoD randomly splits it into two intervals and finds out that its right interval still contains more than 4 individuals. SoD recursively calls `Split` to split that interval. By conducting the recursive split operation until no more interval has to be split, 4 splits make the value range 5 intervals. Moreover, in Figure 4(c) the population is at generation 20, and the split rate is decreased to 0.3. SoD runs on the population, and the value range is split into 6 regions by 5 split points.

One might wonder that the proposed encoding scheme seems similar to the marginal fixed-height histogram (FHH) introduced in [16]. In fact, there are two significant differences between SoD and FHH. The first difference is the size of the code table. In FHH, the height of the histogram is fixed, and for any population, the number of bins employed in the algorithm is fixed. However, in SoD, even with the same split rate, for different populations, SoD may generate the code table of different sizes. That is, the code table size in SoD may vary. For the other difference, the MPM model built according to the individuals encoded by SoD is not of the identical height. Such a flexibility might make the MPM model more accurate than that built according to the individuals encoded by FHH.

For handling the adaptive discretization during an optimization process, Figure 5 shows an example of how SoD cooperating with ECGA splits the solution space at different generations when minimizing a two-dimensional objective function $F_1 = \sum x_i^2$, where the bound of every dimension is $[-100, 100]$, and the global optimum is $(0, 0)$. Figure 5(a) depicts the split configuration on the solution space at generation 1. The split configuration seems random because the whole population is highly diverse at generation 1. Later on, at generation 50, the population begins to converge, and Figure 5(b) shows that SoD splits the solution space around $(0, 0)$ into many regions and leaves other parts of the solution space unencoded. Finally, in Figure 5(c), it can be observed that SoD focuses on the solution space close to $(0, 0)$ at generation 100. With the population converging to $(0, 0)$, ECGA is able to explore the promising solution space more thoroughly and to find the solutions of the higher precision with the assistance of SoD.

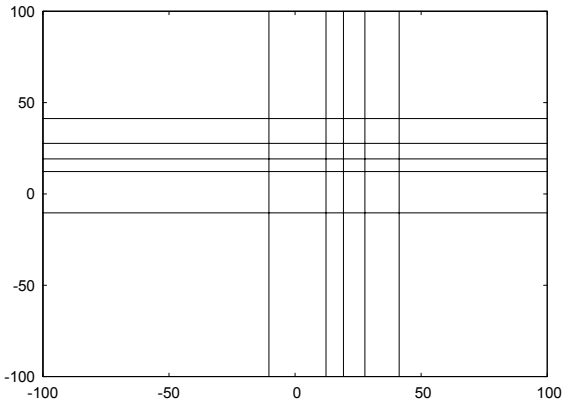
Another example is the two-dimensional objective function $F_2 = \sum 10 - |x_i|$. Figure 6 depicts how SoD splits the solution space, of which the bound of each dimension is $[-10, 10]$, when minimizing F_2. There are four global minima located at $(-10, -10)$, $(-10, 10)$, $(10, -10)$, and $(10, 10)$, respectively. Figure 6(a) is the split configuration on solution space at generation 1. Because the population is initially random, the split configuration seems random. In Figure 6(b), we can observe that at generation 10, because the population begins to converge to the global minima, the split points are close to the four corners where the global minima of F_2 are located. Finally, Figure 6(c) shows that almost all split points are around the region close to $(10, 10)$ because the population converge to only one of the four global optima at generation 20.

These two examples demonstrate that the split configuration established by SoD appropriately responds to the status of the population. The split configuration can encode the individuals as precise as necessary for the cooperating PMBGA to build probabilistic models. Hence, SoD is an effective encoding scheme to make PMBGAs to tackle the real-parameter optimization problem. In next section, ECGA, as an example of PMBGAs, will be employed to show the feasibility of integrating SoD and PMBGAs.

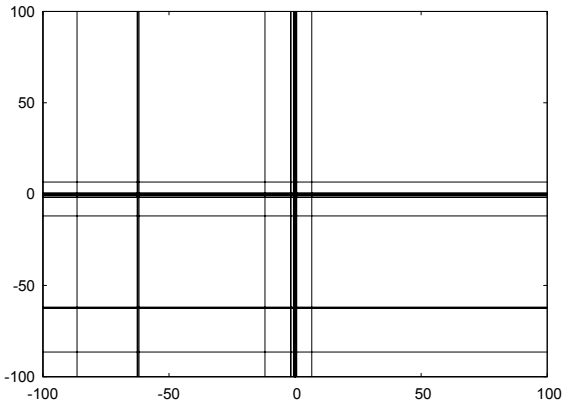
4. rECGA

In the previous sections, we proposed Split-on-Demand, described the behavior of SoD, and demonstrated the effect of SoD. In this section, we will show the way to plug SoD into ECGA, as a showcase for the integration of SoD and PMBGAs. The outcome is a new algorithm, called the *real-coded* ECGA (rECGA), for solving real-parameter optimization problems with the search power provided by ECGA. rECGA can be put as:

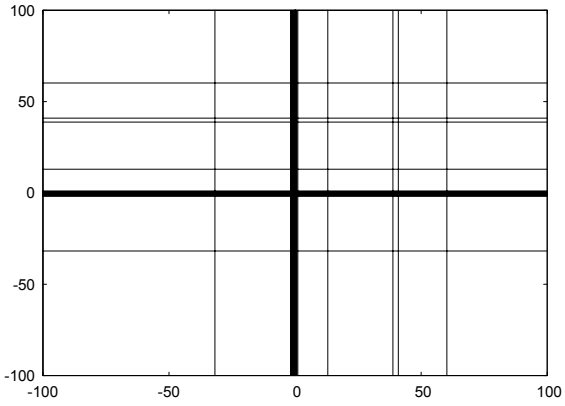
1. Generate a random population of size N.
2. Apply the tournament selection at a rate S.
3. Use SoD to encode each dimension.
4. Model the population using a greedy MPM search.
5. If the model has converged, stop.
6. Generate a new population using the given model.
7. For every L generations,
   (a) Sort the whole population.
   (b) Run the Simplex algorithm on the best 10% individuals.
8. Return to step 2.



(a) Generation 1.

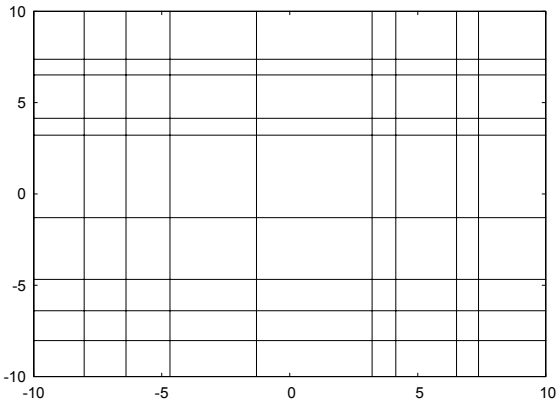


(b) Generation 50.

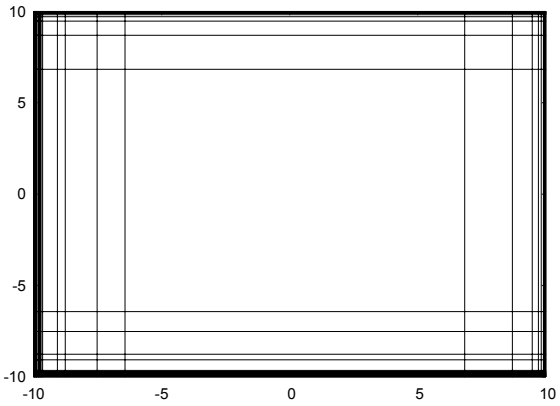


(c) Generation 100.

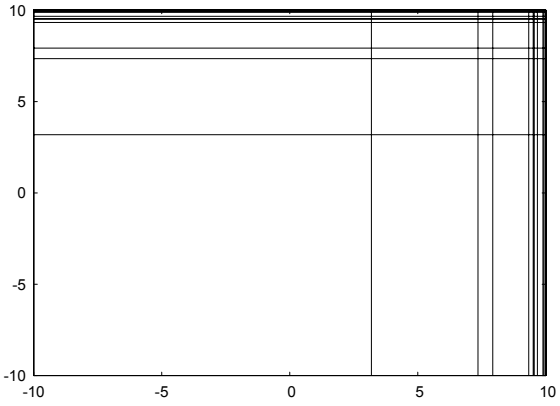
Figure 5: Split configurations at different generations for the objective function $F_1 = \sum x_i^2$.



(a) Generation 1.



(b) Generation 10.



(c) Generation 20.

Figure 6: Split configurations at different generations for the objective function $F_2 = \sum 10 - |x_i|$.

In rECGA, we use SoD to encode each dimension of the individuals in the current population after tournament selection and do the MPM greedy search as in ECGA. We also use a local search method in rECGA to improve the performance. For every L generations, the population is sorted according to the objective values, and the Simplex algorithm proposed by Nelder and Mead [8] is conducted on the best 10% individuals.

5. EXPERIMENTS

In this section, we will use rECGA to solve a set of test functions and show the experimental results. The parameters of rECGA we use in the series of experiments are population size = 250, probability of crossover = 0.975, tournament size = 8, $\gamma = 0.5$, $\epsilon = 0.998$, and $L = 5$.

5.1 Test Functions

There are several optimization methodologies designed to solve real-parameter optimization problems. The popular approaches include real-parameter EAs, evolution strategies (ES), differential evolution (DE), particle swarm optimization (PSO), classic mathematical methods, such as quasi-Newton method (QN), and hybridization of evolutionary algorithms and classic methods. These methodologies are quite different from each other in their operators, concepts, and working principles. In order to make fair comparisons between these optimization methods, a set of benchmark functions for testing real-parameter optimization algorithms was proposed in the CEC 2005 [15] as an attempt to setup a standard set of benchmark functions of different properties and characteristics.

In addition to the set of real-parameter optimization benchmark functions, the special session on real-parameter optimization at CEC 2005 also established the evaluation criteria as well as provided the performance results of many optimization methodologies, including those aforementioned algorithms, for comparison. Therefore, the performance of rECGA will be compared to that of the existing algorithms included in the special session at CEC 2005.

5.2 Experimental Results

The error values, $f(x) - f(x^*)$, described in [15], are presented in Tables 1–5 for the 25 test functions. Because of the space limitation, we compare the results of rECGA with those of the population-based, steady-state optimization algorithm [14], which was publiched in the special session. The error values are also presented in the same format in Table 6–10. Each column of the table corresponds to one test function, and the number of dimensions for each problem is 10. The error values are recorded after $1,000$, $10,000$, and $100,000$ function evaluations (FEs) for each one of the 25 runs. A run is considered a *success* if the final solution reaches within the given fixed accuracy level. The predefined accuracy levels are $1e-6$ for functions 1–5, $1e-2$ for functions 6–14, and $1e-1$ for functions 15–25. The error values of the 25 runs on one function are sorted and the tables present the following items: the 1st (Best), the 13th (Median), the 25th (Worst), and the average (Mean). The tags † and ‡ put after the function number denote that the function is considered solved or rECGA obtains comparable results against other advanced algorithms, respectively. If rECGA successfully reaches within the given accuracy level for the particular function in at least one out of the 25 runs, the function is considered solved by rECGA. Moreover, by comparable results, we mean that rECGA does not reach the given accuracy as other advanced algorithms and the performance of rECGA is equally good compared to that of other algorithms.

Function		1†	2†	3†	4†	5‡
1e3	Best	1.30e+03	2.79e+03	1.07e+07	4.37e+03	3.70e+03
	Median	2.31e+03	5.76e+03	4.13e+07	8.14e+03	6.86e+03
	Worst	3.62e+03	1.22e+04	9.05e+07	1.45e+04	9.03e+03
	Mean	2.34e+03	6.40e+03	4.33e+07	8.33e+03	6.77e+03
1e4	Best	3.71e-02	1.31e+01	9.15e+04	2.37e+01	5.43e+00
	Median	2.42e-01	5.81e+01	1.01e+06	7.49e+01	4.22e+01
	Worst	1.39e+00	1.52e+02	3.52e+06	3.08e+02	2.21e+02
	Mean	3.37e-01	5.92e+01	1.22e+06	1.03e+02	6.24e+01
1e5	Best	5.68e-14	1.14e-13	1.14e-13	6.06e-08	1.23e-04
	Median	5.68e-13	1.08e-12	2.16e-12	5.98e-05	7.44e-04
	Worst	2.33e-12	2.26e-11	2.58e+02	1.40e-02	5.48e+00
	Mean	7.16e-13	2.37e-12	1.03e+01	1.25e-03	3.78e-01

† Considered solved according to the given accuracy.
‡ Comparable to the results obtained by other algorithms.

Table 1: Error values for function 1–5.

Function		6†	7†	8‡	9†	10
1e3	Best	3.58e+07	1.42e+03	2.05e+01	4.87e+01	5.95e+01
	Median	1.08e+08	1.64e+03	2.08e+01	6.78e+01	8.52e+01
	Worst	3.98e+08	1.83e+03	2.09e+01	8.12e+01	1.05e+02
	Mean	1.19e+08	1.66e+03	2.08e+01	6.66e+01	8.53e+01
1e4	Best	8.83e+01	1.23e+03	2.04e+01	3.75e-01	9.32e+00
	Median	4.06e+02	1.24e+03	2.05e+01	5.46e+00	1.47e+01
	Worst	3.41e+03	1.25e+03	2.06e+01	1.13e+01	3.49e+01
	Mean	8.00e+02	1.24e+03	2.05e+01	5.65e+00	1.82e+01
1e5	Best	3.41e-13	9.86e-03	2.00e+01	1.14e-13	4.98e+00
	Median	3.99e+00	2.73e-01	2.00e+01	1.42e-12	1.29e+01
	Worst	9.87e+01	5.07e+00	2.00e+01	2.87e-11	3.08e+01
	Mean	1.03e+01	5.27e-01	2.00e+01	3.64e-12	1.31e+01

† Considered solved according to the given accuracy.
‡ Comparable to the results obtained by other algorithms.

Table 2: Error values for functions 6–10.

Function		11‡	12†	13‡	14	15†
1e3	Best	9.93e+00	1.43e+04	4.26e+01	3.91e+00	5.80e+02
	Median	1.20e+01	3.26e+04	3.73e+02	4.29e+00	6.93e+02
	Worst	1.35e+01	5.15e+04	3.31e+03	4.53e+00	7.56e+02
	Mean	1.19e+01	3.41e+04	6.22e+02	4.27e+00	6.90e+02
1e4	Best	2.93e+00	1.50e+02	3.54e-01	2.99e+00	8.06e+01
	Median	5.52e+00	7.71e+02	1.98e+00	3.53e+00	4.53e+02
	Worst	9.16e+00	4.40e+03	3.42e+00	4.07e+00	5.13e+02
	Mean	5.21e+00	1.10e+03	1.95e+00	3.49e+00	3.92e+02
1e5	Best	1.21e+00	1.71e-13	4.94e-02	1.79e+00	1.85e-13
	Median	3.84e+00	1.19e-11	4.31e-01	3.07e+00	4.28e+02
	Worst	7.82e+00	1.69e+03	1.06e+00	4.02e+00	4.42e+02
	Mean	3.85e+00	2.23e+02	4.56e-01	3.12e+00	3.08e+02

† Considered solved according to the given accuracy.
‡ Comparable to the results obtained by other algorithms.

Table 3: Error values for functions 11–15.

The experimental results indicate that rECGA can solve the functions 1, 2, 3, 4, 6, 7, 9, 12, and 15, which are denoted with †. Functions 1 and 2 are simple problems and can be solved in every run. Function 3 is the shifted rotated high conditioned elliptic function, which magnifies the error of input. Even if the error of input is quite small, the error value will be huge due to a huge multiplier. By utilizing the good local search operator, rECGA is able to solve function

Function		16	17	18	19	20‡
1e3	Best	2.17e+02	2.88e+02	1.07e+03	1.01e+03	1.08e+03
	Median	3.30e+02	3.61e+02	1.12e+03	1.10e+03	1.12e+03
	Worst	4.13e+02	4.69e+02	1.16e+03	1.16e+03	1.16e+03
	Mean	3.22e+02	3.62e+02	1.12e+03	1.10e+03	1.12e+03
1e4	Best	1.01e+02	1.17e+02	4.15e+02	3.83e+02	4.48e+02
	Median	1.29e+02	1.58e+02	9.17e+02	8.01e+02	8.93e+02
	Worst	1.90e+02	2.21e+02	1.01e+03	9.69e+02	1.02e+03
	Mean	1.30e+02	1.60e+02	8.34e+02	8.14e+02	8.36e+02
1e5	Best	9.87e+01	1.08e+02	3.00e+02	3.00e+02	3.00e+02
	Median	1.23e+02	1.29e+02	9.08e+02	8.00e+02	8.88e+02
	Worst	1.55e+02	1.51e+02	1.00e+03	9.64e+02	1.01e+03
	Mean	1.22e+02	1.30e+02	7.79e+02	7.95e+02	7.73e+02

† Considered solved according to the given accuracy.
‡ Comparable to the results obtained by other algorithms.

Table 4: Error values for function 16–20.

Function		21‡	22	23‡	24‡	25
1e3	Best	1.14e+03	9.64e+02	1.02e+03	9.34e+02	1.82e+03
	Median	1.30e+03	1.01e+03	1.31e+03	1.19e+03	1.86e+03
	Worst	1.35e+03	1.08e+03	1.35e+03	1.34e+03	1.89e+03
	Mean	1.28e+03	1.02e+03	1.29e+03	1.17e+03	1.86e+03
1e4	Best	5.00e+02	7.75e+02	5.60e+02	2.00e+02	1.69e+03
	Median	8.50e+02	7.87e+02	5.60e+02	2.01e+02	1.75e+03
	Worst	1.14e+03	9.05e+02	1.25e+03	2.05e+02	1.78e+03
	Mean	7.72e+02	7.96e+02	8.48e+02	2.02e+02	1.75e+03
1e5	Best	3.00e+02	7.32e+02	5.60e+02	2.00e+02	1.49e+03
	Median	5.00e+02	7.59e+02	5.60e+02	2.00e+02	1.73e+03
	Worst	1.13e+03	8.75e+02	1.25e+03	2.00e+02	1.75e+03
	Mean	7.25e+02	7.69e+02	8.21e+02	2.00e+02	1.71e+03

† Considered solved according to the given accuracy.
‡ Comparable to the results obtained by other algorithms.

Table 5: Error values for functions 21–25.

3. Function 4 is the shifted Schwefel's problem 1.2 with noise in fitness. rECGA can solve this problem in that SoD can decrease the noise effect by randomly splitting the real number interval initially. Function 5 is Schwefel's problem 2.6 with global optimum on bounds. The special property of function 5 is that function 5 can be solved if the individuals are at bounds. Because of the property and behavior of SoD, rECGA fails to achieve the success criterion, although rECGA is able to provide comparable results.

Functions 6–14 are basic multimodal problems and expanded multimodal problems. Although rECGA cannot solve all these problems, most of the results are comparable to that of other advanced algorithms. The optimum of function 8 is within a very narrow valley, and the parameter setting used in this experiment does not allow rECGA to have a sufficient resolution to accurately find the valley. Functions 15–25 are composition functions. They are composites of the basic functions, and they are big challenges to search algorithms. rECGA successfully solves only function 15. Some of the results of rECGA for functions 16–25 are comparable to other algorithms. rECGA and many of other algorithms can only find the local optima.

Several difficulties remain to be overcome, and therefore, we will continue to work on SoD to provide a versatile encoding scheme. The future work for SoD includes the following items. (1) SoD can quickly focus on the intervals with

Function		1	2	3	4	5
1e3	Best	3.67e+03	6.26e+03	2.23e+07	9.72e+03	7.30e+03
	Median	8.29e+03	1.33e+04	9.42e+07	1.79e+04	1.110e+04
	Worst	1.10e+04	2.06e+04	1.50e+08	2.91e+04	1.34e+04
	Mean	7.84e+03	1.31e+04	9.31e+07	1.81e+04	1.07e+04
1e4	Best	1.43e-02	7.01e+01	7.75e+06	1.42e+02	1.41e+03
	Median	7.53e-02	2.05e+02	2.50e+07	3.84e+02	2.43e+03
	Worst	1.83e-01	4.33e+02	5.12e+07	8.57e+02	3.28e+03
	Mean	9.42e-02	2.33e+02	2.57e+07	4.08e+02	2.46e+03
1e5	Best	3.76e-09T	7.57e-09T	5.90e+03	8.68e-09T	9.15e-01
	Median	8.31e-09T	9.54e-09T	2.04e+04	9.96e-09T	3.82e+01
	Worst	9.89e-09T	9.88e-09T	9.87e+04	8.02e-06	2.55e+02
	Mean	8.71e-09T	9.40e-09T	3.02e+04	7.94e-07	4.85e+01

Table 6: Error values for functions 1–5 for [14].

Function		6	7	8	9	10
1e3	Best	7.72e+08	4.59e+02	2.04e+01	6.58e+01	8.47e+01
	Median	1.53e+09	7.59e+02	2.07e+01	9.60e+01	1.31e+02
	Worst	3.97e+09	1.26e+03	2.09e+01	1.13e+02	1.56e+02
	Mean	1.82e+09	8.12e+02	2.07e+01	9.49e+01	1.28e+02
1e4	Best	1.77e+05	8.82e+00	2.00e+01	3.28e+01	4.12e+01
	Median	1.12e+06	1.71e+01	2.00e+01	4.55e+01	5.30e+01
	Worst	4.94e+06	2.36e+01	2.02e+01	5.72e+01	6.58e+01
	Mean	1.24e+06	1.70e+01	2.00e+01	4.51e+01	5.25e+01
1e5	Best	8.30e-09T	5.27e-09T	2.00e+01	4.76e-09T	6.26e-09T
	Median	3.99e+00	5.42e-02	2.00e+01	8.82e-09T	8.85e-09T
	Worst	1.09e+02	1.53e-01	2.00e+01	2.98e+00	2.98e+00
	Mean	2.07e+01	6.40e-02	2.00e+01	1.19e-01	2.39e-01

Table 7: Error values for functions 6–10 for [14].

Function		11	12	13	14	15
1e3	Best	9.42e+00	2.56e+04	8.93e+01	4.13e+00	7.38e+02
	Median	1.16e+01	7.60e+04	1.96e+03	4.38e+00	8.50e+02
	Worst	1.29e+01	1.32e+05	9.84e+03	4.63e+00	9.10e+02
	Mean	1.14e+01	8.29e+04	2.38e+03	4.39e+00	8.41e+02
1e4	Best	7.63e+00	2.56e+04	2.59e+00	3.67e+00	6.37e+02
	Median	1.01e+01	4.55e+04	3.53e+00	4.27e+00	7.34e+02
	Worst	1.18e+01	7.14e+04	4.66e+00	4.51e+00	7.88e+02
	Mean	1.01e+01	4.78e+04	3.63e+00	4.25e+00	7.25e+02
1e5	Best	7.64e+00	1.98e+02	3.28e-01	1.53e+00	2.79e+02
	Median	9.22e+00	2.50e+04	7.15e-01	2.40e+00	4.63e+02
	Worst	9.91e+00	4.00e+04	1.07e+00	3.30e+00	7.06e+02
	Mean	9.11e+00	2.44e+04	6.53e-01	2.35e+00	5.10e+02

Table 8: Error values for functions 11–15 for [14].

many individuals. However, it might ignore some intervals with fewer individuals. If we use the niching techniques to distribute the computation power to more intervals, we may avoid such a problem caused by many local optima. (2) By using the fixed coordinate system, it cannot model the solution space accurately when the problem has the rotated properties. In this case, rotating the coordinate system with the problem might be a possible way to improve the performance of SoD. (3) In our current framework, we use the uniform distribution to spilt the solution space. We might try to use other probability distributions such that the split configuration of the solution space might provide further search power for SoD. (4) We do not know very well

Function		16	17	18	19	20
1e3	Best	3.78e+02	4.07e+02	1.09e+03	1.11e+03	1.06e+03
	Median	4.35e+02	5.25e+02	1.22e+03	1.23e+03	1.20e+03
	Worst	5.59e+02	6.67e+02	1.30e+03	1.32e+03	1.34e+03
	Mean	4.47e+02	5.27e+02	1.22e+03	1.22e+03	1.21e+03
1e4	Best	1.73e+02	1.72e+02	8.48e+02	8.47e+02	8.60e+02
	Median	1.95e+02	2.17e+02	9.47e+02	9.91e+02	9.77e+02
	Worst	2.22e+02	2.42e+02	1.06e+03	1.08e+03	1.07e+03
	Mean	1.95e+02	2.14e+02	9.52e+02	9.76e+02	9.74e+02
1e5	Best	8.75e+01	8.82e+01	3.00e+02	3.00e+02	3.00e+02
	Median	9.37e+01	9.69e+01	8.22e+02	8.24e+02	8.26e+02
	Worst	1.13e+02	1.14e+02	9.51e+02	9.42e+02	9.59e+02
	Mean	9.59e+01	9.73e+01	7.52e+02	7.51e+02	8.13e+02

Table 9: Error values for functions 16–20 for [14].

Function		21	22	23	24	25
1e3	Best	1.22e+03	9.37e+02	1.25e+03	4.86e+02	5.53e+02
	Median	1.35e+03	1.10e+03	1.41e+03	7.22e+02	7.73e+02
	Worst	1.40e+03	1.22e+03	1.45e+03	9.95e+02	9.74e+02
	Mean	1.34e+03	1.09e+03	1.40e+03	7.63e+02	7.72e+02
1e4	Best	9.42e+02	5.50e+02	1.03e+03	4.08e+02	4.10e+02
	Median	1.13e+03	7.95e+02	1.17e+03	4.12e+02	4.11e+02
	Worst	1.14e+03	9.30e+02	1.20e+03	4.13e+02	4.13e+02
	Mean	1.12e+03	7.43e+02	1.16e+03	4.11e+02	4.11e+02
1e5	Best	5.00e+02	5.27e+02	5.59e+02	4.05e+02	4.05e+02
	Median	1.08e+03	7.29e+02	1.10e+03	4.06e+02	4.06e+02
	Worst	1.09e+03	8.69e+02	1.11e+03	4.07e+02	4.07e+02
	Mean	1.05e+03	6.59e+02	1.06e+03	4.06e+02	4.06e+02

Table 10: Error values for functions 21–25 for [14].

about the relationship between SoD's parameters and its behavior, it needs more study. (5) Finally, the integration of SoD and ECGA works well on the standard benchmark functions. Application of SoD to other PMBGAs should be investigated in the future.

6. CONCLUSIONS

In the present work, we proposed an adaptive discretization method, called *Split-on-Demand* (SoD), to enable the PMBGAs or EDAs designed for handling bit-strings to tackle real-parameter optimization problems. The procedure of SoD was presented in detail, and the effect of SoD was displayed. As an example, we also demonstrated the way to integrate SoD into ECGA, named *real-coded* ECGA (rECGA), and examined rECGA with a recently defined set of benchmark functions. The experimental results were compared to that of other advanced methodologies and indicated that rECGA work well on the set of test functions. After discussing the performance of rECGA on different test functions, the future work to enhance SoD was presented.

SoD is designed and developed for adaptively discretizing real number intervals to assist PMBGAs, EDAs, and other algorithms to work on real numbers. SoD reflects the distribution of the current population and encodes the real numbers in discrete codes as necessary. The usage of SoD makes it easy to be applied to the existing algorithms designed for the bit or integer representations. This paper shows that SoD is simple and flexible, and the numerical results of the

example rECGA indicate that developing SoD is a promising research direction. More work along this line needs to be done. We will continue to work on SoD to enhance and improve its capability and applicability.

7. ACKNOWLEDGMENTS

The work was partially sponsored by the National Science Council of Taiwan under grant NSC-94-2213-E-009-120. The authors are also grateful to the National Center for High-performance Computing for computer time and facilities.

8. REFERENCES

[1] C. W. Ahn, R. S. Ramakrishna, and D. E. Goldberg. Real-coded Bayesian optimization algorithm, bringing the strength of BOA into the continuous world. In *Proceedings of the GECCO 2004 Genetic and Evolutionary Computation Conference*, pages 840–851, 2004.

[2] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, New York, 1989.

[3] D. E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*, volume 7 of *Genetic Algorithms and Evoluationary Computation*. Kluwer Academic Publishers, June 2002. ISBN: 1-4020-7098-5.

[4] D. E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530, 1989. (Also TCGA Report No. 89003).

[5] G. R. Harik. Linkage learning via probabilistic modeling in the ECGA. IlliGAL Report No. 99010, University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory, Urbana, IL, 1999.

[6] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975. ISBN: 0-262-58111-6.

[7] P. Larrañaga and J. A. Lozano. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*, volume 2 of *Genetic algorithms and evolutionary computation*. Kluwer Academic Publishers, Boston, MA, October 2001. ISBN: 0-7923-7466-5.

[8] J. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–315, 1965.

[9] M. Pelikan, D. E. Goldberg, and F. G. Lobo. A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications*, 21(1):5–20, 2002. (Also IlliGAL Report No. 99018).

[10] J. Rissanen. *Stochastic Complexity in Statistical Inquiry*. World Science, 1989.

[11] M. Sebag and A. Ducoulombier. Extending population-based incremental learning to continuous search spaces. In *Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature (PPSN V)*, pages 418–427, 1998.

[12] I. L. Servet, L. Trave-Massuyes, and D. Stern. Telephone network traffic overloading diagnosis and evolutionary computation techniques. In *Proceeings of the Third European Conference on Artificial Evolution (AE 97)*, pages 137–144, 1997.

[13] S.-Y. Shin and B.-T. Zhang. Bayesian evolutionary algorithms for continuous function optimization. In *Proceedings of the 2001 Congress on Evolutionary Computation (CEC2001)*, pages 508–515, 2001.

[14] A. Sinha, S. Tiwari, and K. Deb. A population-based, steady-state procedure for real-parameter optimization. In *Proceedings of the 2005 Congress on Evolutionary Computation (CEC2005)*, pages 514–521, 2005.

[15] P. N. Suganthan, N. Hansen, J. J. Liang, K. Deb, Y.-p. Chen, A. Auger, and S. Tiwari. Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization. Technical Report, Nanyang Technological University, Singapore, May 2005.

[16] S. Tsutsui, M. Pelikan, and D. E. Goldberg. Evolutionary algorithm using marginal histogram models in continuous domain. IlliGAL Report No. 2001019, University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory, Urbana, IL, 2001.