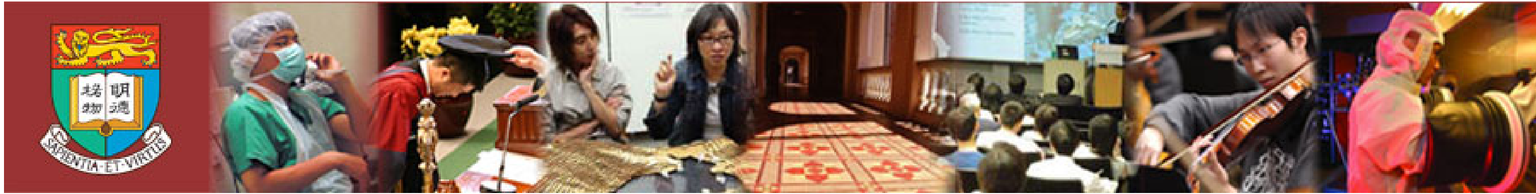
CORE
Metadata, citation and similar papers at core.ac.uk
Provided by HKU Scholars Hub



Title	A non-revisiting genetic algorithm
Author(s)	Yuen, SY; Chow, CK
Citation	The 2007 IEEE Congress on Evolutionary Computation (CEC 2007), Singapore, 25-28 September 2007. In IEEE Transactions on Evolutionary Computation, 2007, p. 4583-4590
Issued Date	2007
URL	http://hdl.handle.net/10722/196701
Rights	Creative Commons: Attribution 3.0 Hong Kong License

A Non-Revisiting Genetic Algorithm

Shiu Yin Yuen Chi Kin Chow
Department of Electronic Engineering
City University of Hong Kong
Hong Kong, China
{kelviny.ee, chowchi}@cityu.edu.hk

Abstract - **Genetic Algorithm (GA) is a revisiting stochastic algorithm. In other words, a solution that has been visited before may be revisited. The fitness of the solution has to be evaluated each time. Since fitness evaluation is the most computationally intensive process in the execution of the GA, revisits should be minimized or eliminated. In this paper, a novel dynamic binary partitioning tree archive is proposed to eliminate all revisits. It works as follows: When the GA generates a solution, the tree is accessed. A leaf node is appended to the tree if the solution has not been visited before and so has no record in the tree. Otherwise, a search is initiated from the leaf node that is the duplicate to the solution to find the nearest neighbor solution in the search space that is not visited. During this process, whole sub-trees may be pruned if all the leaf nodes it contains are visited. The search naturally implements a self adaptive mutation mechanism. Hence the GA requires no other mutation parameter or mutation scheme. Experimental results reveal that this new GA is superior in performance compared with the standard GA with revisits, and the tree archive is not memory intensive.**

I. Introduction

Many stochastic algorithms (e.g. Genetic Algorithm, Simulated Annealing, etc) do not memorize places that they have visited. Exceptions are Tabu search [1] and Particle Swarm Optimization [2], which incorporates the memory of recent search results to guide the next search step. However, even with these algorithms, not the entire list of visited positions is recorded. Thus, revisits are inevitable. A revisit to a search position s is defined as a re-evaluation of the function (or fitness) of s which has been evaluated before. Since function evaluation is usually the most computational intensive process within the stochastic algorithm, such revisits are clearly wasteful of computational resources. In the terminology of the No-free-lunch (NFL) theorems [3], all non-revisiting (stochastic or deterministic) algorithms have the same average performance when the problem distribution is uniform. A revisiting algorithm A searches the same sequence of distinct points as a non-revisiting algorithm A' when revisited points in the sequence are taken out. This implies that A' is superior to A as A' has the same performance but with fewer function evaluations. Thus it is always beneficial to eliminate all the revisits.

A naive way to eliminate the revisits is to keep an archive of all visited points so far. Unfortunately, if the search space is of size a^{γ}, where a is the cardinality of the alphabet, then the worst case memory complexity of the archive is a^{γ} - 1. This is infeasible for any realistic search space size. In this paper, we propose a practical solution to the revisit problem in the context of the Genetic Algorithm (GA). We show that an archive design can be naturally integrated with the GA so that revisits are completely eliminated while the archive size is kept reasonably small and practically feasible (see section III: Experimental Results). A modification of the GA by incorporating an adaptive mutation naturally arises from the integration.

The GA is a famous stochastic optimization algorithm pioneered by Holland [4]. It mimics the evolutionary process of a population of individuals over time. The hall mark of a GA is the provision of a population (multiple parallel search capability), selection (survival of the fittest), crossover (sexual reproduction) and mutation (random incremental changes). The GA is closely related to the Evolutionary Strategies (ES) and Evolutionary Programming (EP). There are many flavors in the design of GA, for example, there exists many different designs for the population structure, selection, mutation, and crossover [5]. Our modification of the GA is applicable to both discrete and continuous space problems.

It is recognized very early on that duplicate removal can enhance the performance of the GA significantly. Mauldin [6]'s uniqueness operator only allow a new child to be inserted into the population if its Hamming distance to all members of the population is greater than a threshold. Davis [7] reports that a binary coded GA that removes duplicates in the population results in superior performance in a comparable number of child evaluations. These researches compare each child with each solution in the current population. For a population with δ individuals, δ comparisons need to be made. Ronald [8] reported the use of Hash table to reduce the number of comparison to $O(1)$. However, these efforts only compare

1-4244-1340-0/07$25.00 ©2007 IEEE

a child with the current population and do not guarantee no revisits in the entire search.

From another angle, it is generally agreed by the GA research community that to prevent premature convergence, an appropriate diversity in the population has to be maintained. The reason is that once the entire population converges to a single kind of individual, crossover will be useless and the GA reduces to parallel mutation climbing. Numerous operators that modify the selection, crossover or mutation to diversify the population have been proposed. Some more famous examples are fitness sharing [9], rank based selection [10], elitist selection with incest avoiding [11] and crowding [12]. Our method automatically guarantees that each individual in a current population is different from each other without the need to introduce any special operator.

From yet another angle, it is known from theoretical studies [13, 14] that an adaptive mutation rate is beneficial for the GA to find the global optimum more efficiently. However, researchers still have little idea on how to design a suitable adaptive mutation schedule for general optimization problems, see survey by [15]. This paper offers a fresh perspective to this problem. Our method naturally suggests *the* adaptive mutation mechanism and requires no tuning parameters. Our GA does not need to set any mutation parameter or decide on the mutation scheme. It would automatically increase its mutation rate if it is too small, and there is a physical meaning to the adjective "small" in the GA - The mutation must be big enough to find the nearest unvisited neighbor.

The use of a dynamic archive in single objective optimization using GA is novel to our knowledge. The use of archive in the existing GA literature is limited to the use of adaptive archive in MultiObjective EA (MOEA). There the problem is to use an archive to store the best solutions that are pareto optimal, as in MOEA the pareto optimal set can be sizable [16]. The problem is completely different and bears no relation to this research.

This paper is organized as follows: Section II reports the GA with the dynamic tree archive method. Section III reports the experimental results. Section IV gives the conclusion.

II. Genetic Algorithm with Dynamic Tree Archive

A binary space partitioning (BSP) tree archive *Ar* is constructed. A GA can be visualized as generating a sequence of solutions **sq** = ($s(1)$, $s(2)$, …). For a generational GA with population size δ, δ solutions are generated in the same cycle. For a steady state GA, solutions are generated one by one until δ solutions are generated. Both can be considered as generating a sequence of solutions. The function of *Ar* is to return an amended sequence **sq'** = ($s(1)'$, $s(2)'$, …) such that no two $s(i)'$ and $s(j)'$ satisfy $s(i)' = s(j)'$ unless $i = j$. For clarity, let ignore revisits for the moment. That is, $s(i) \neq s(j)$ unless $i = j$. Then each solution $\mathbf{x} = (x_1, \ldots, x_\gamma)$ ($\mathbf{x}$ can be $s(1)$, $s(2)$, etc) will insert a leaf node $\mathbf{x}$ to the tree. x_j are variables with cardinality a. We need the following metric to partition the search space:

Definition 1: Metric $d(\mathbf{x},\mathbf{y} \mid j)$

The Euclidean distance between x_j and y_j is designated as the metric $d(\mathbf{x},\mathbf{y} \mid j)$.

Remark: Other metric, for example the Hamming, may also be employed.

The following algorithm constructs the archive *Ar* when there are no revisits:

Algorithm A1: (Archive construction when there are no re-visits)

```
1. Initial condition: Ar consists of the root
   node
2. A new solution z (generated by the GA) is
   presented to Ar
3. Curr_node := root
4. If (Curr_node has two child nodes)
   {
      Compare z with child node x and y;
      Define the comparing dimension j:
              j = arg max_{k∈[1,γ]} d(x,y | k)
      If d(x,z | j) ≤ d(y,z | j)
              Curr_node := child node x
      Else
              Curr_node := child node y
      Repeat (Step 4)
   }
   Else
   {
     Insert a child node (that records) z to
      Curr_node*
      Finish
   }
```

Remark: If the objective function is a pseudo boolean function, the comparing dimension j defined in step 4 is randomly chosen subject to the condition: $x_j \neq y_j$.

The archive is a standard BSP tree with the following well established properties:

1. The root represents the entire search space S.
2. Each non-root node represents a subspace of S. Precisely, it is a hyper-rectangular box in the search space.
3. Suppose a parent node has two child nodes **x** and **y**, then the subspaces of **x** and **y** are disjoint and their

union is the subspace of the parent, i.e., the child nodes binary partition the parent subspace.

Remark:* If *Curr_node* in the algorithm *A1* represents a subspace that has only a single point, then no insertion is performed. (It will be marked *Closed*. See *Definition 3* below.)

The following properties are specific to the BSP tree proposed for GA:

1. It is constructed dynamically using the data points generated by the GA. Thus, each run of GA will produce a different tree.
2. It stores all the search positions **x** visited by the GA. (Recall that we make the simplifying assumption that there are no revisits for the moment.)
3. For a balanced tree, the mean number of steps to decide whether a search position **x** has been visited is at most $O(\log(a^\gamma))$.

Next, consider the general case when revisits could occur:

Definition 2: Revisits

The ith solution $s(i)$ in the sequence $\mathbf{sq} = (s(1), s(2), \ldots)$ is a revisit if $\exists\, j < i,\ s(i) = s(j)$. It occurs if and only if *Curr_node* $= s(i)$ is found when traversing the tree.

Recall that each node in *Ar* records a solution in the set $\{s(1), \ldots, s(i-1)\}$. Thus a revisit may occur when Algorithm *A1* is visiting a non-leaf (intermediate) node or a leaf node.

Before introducing the algorithm that handles revisits, it is desirable to enlarge the utility of the concept that *Curr_node* = **x** in the following way, i.e. this information may also be interpreted as *Curr_node* recording the *entire subspace* under **x**. Denote the subspace spanned by **x** as *X*. In the tree, it is represented by introducing a status flag:

Definition 3: Open and Closed node

A node **x** is flagged *Open* if the subspace represented by **x** has not all been visited. Otherwise, it is flagged *Closed*.

Remark: By definition, a node **x** for which the subspace contains a single point is always flagged *Closed*, since the single point, which simultaneously is the whole subspace, has been visited at the time when the node is created.

Thus when the tree traversal visits a *Closed* node, then a revisit has occurred.

The following algorithm deals with revisits. The newly added codes are in bold:

Algorithm A2 (Archive construction allowing revisits)

```
1. Initial condition: Ar consists of the root
   node. Flag(root) := Open
2. A new solution z (generated by the GA) is
   presented to Ar
3. Curr_node := root
4. Case 1: Flag(Curr_node) = Open
   If (Curr_node has two child nodes)
   {
      Compare z with child node x and y;
      Define the comparing dimension j:
            j = arg max_{k∈[1,γ]} d(x,y|k)
     If d(x,z | j) ≤ d(y,z | j)
            Curr_node := child node x
     Else
            Curr_node := child node y
     Repeat (Step 4)
   }
   Else
   {
     Insert a child node (that records) z to
     Curr_node
     If (child node's subspace is not a
     singleton)
          Flag(child node) := Open
     Else
          Flag(child node) := Closed
     Finish
   }

   Case 2: Flag(Curr_node) = Closed
   Curr_node := Parent
   If (there are two Closed child nodes)     //
   Case 1
   {
     Flag(Curr_node) := Closed
      Prune the subtree under Curr_node
      Repeat (Step 4)
   }
   Else If (there is an Open child node)     //
   Case 2
   {
      Curr_node := Open child node
      Repeat (Step 4)
   }
   Else // Case 3
   {
      Evaluate the unvisited subspace
      represented by Curr_node
      Create a child node by mutating z within
      the unvisited subspace
      If (child node's subspace is not a
      singleton)
          Flag(child node) := Open
      Else
```

```
            Flag(child node) := Closed
        Finish
    }
```

Algorithm *A2* is the same as Algorithm *A1* except when revisits are encountered. Consider a node **x** that is closed. This implies that all the points in *X* have been visited. The algorithm backtracks to its parent node **p**. Three cases may occur:

Case 1: Node **y**, the other child node of **p**, exists and is also *Closed*. Then the entire subspace of **p**, $P = X \cup Y$, has been visited. *P* is flagged *Closed* as well. The algorithm backtracks to the parent of **p**. Meanwhile, the entire sub-tree under **p** is pruned.

Case 2: Node **y** exists and is *Open*. Then the entire subspace *X* has been visited, but there are still some unvisited points within $Y = P - X$. Then the search is directed to node **y**.

Case 3: Node **y** does not exist. Then the entire subspace $P - X$ has not been visited. This forms the unvisited subspace. Now mutate **z** to some point **z'** within $P - X$. **z'** is a point in $P - X$ that has the minimum distance to **z**, i.e.,

$$\mathbf{z}' = \arg \min_{\mathbf{a} \in P-X} \| \mathbf{a} - \mathbf{z} \|$$

where $\|.\|$ denotes the Euclidean distance.

When there is more than one point that has the same minimum distance, one is arbitrarily chosen.

The main idea for handling revisits is as follows: When a node that stores the solution is identical to the solution generated by the GA, **z**, is found, a revisit has occurred. The tree needs to generate a solution that is not visited before. It does so by backtracking to the parent and check whether the subspace of the parent has all been visited (*Case* 1 above). When this occurs, backtracking is again initiated and the process goes on until the parent's subspace have not been visited entirely (*Case* 2 and 3). Meanwhile, since the whole sub-tree under the parent node has been visited, there is no need to keep the sub-tree. Hence the entire sub-tree is pruned. This pruning is important to keep the size of the tree at any one time small.

Meanwhile, when the parent's subspace has all been visited, the situation can be further subdivided into two cases: *Case* 2 and 3. For case 2, there is an *Open* child node that means that some solutions under the child node's subspace has been visited, but not all. A depth first order search commences to go down the sub-tree of this child node. By nature of the binary space partitioning, it would eventually generate a leaf node (unvisited solution) in this sub-tree that has the minimum distance to *z*. Thus it is implementing a *nearest neighbor search*.

For case 3, the location of the subspace that is unvisited that is the nearest neighbor(s) to **z** has finally been found. To generate an unvisited leaf node, **z** is "mutated" to the nearest neighbor solution in this unvisited subspace. In the degenerate case where there is more than one nearest neighbor, one is arbitrarily chosen.

Observe that the position **z'** that **z** will mutate to is a function of **z** and $P - X$. It is not a specific mutation with a fixed mutation rate (cf. 1 point, 2 point, or uniform). However, it will mutate to a point in the neighborhood of **z**. Also, though the mutation *per se* is deterministic, **z** and *X* are both random and their behaviour depends on the problem. Thus the mutation process when taken as a whole is also random. This agrees with the intuitive meaning of the term mutation. Loosely speaking, when the searched positions in the neighborhood of **z** is sparse, the mutation jumps to nearby points. When the neighborhood is dense, the mutation jumps to points further away. It is also problem specific.

Note that no other mutation operator needs to be defined within the GA. After selection and crossover, the solution can be passed immediately to the tree. If the solution has not been visited, no mutation will be performed. Otherwise, it will mutate to the nearest unvisited neighbor. Note that we have introduced a novel adaptive mutation operator that has the desirable property of having no parameter that requires user fine tuning, i.e. a parameter-less mutation operator.

The efficacy of such a BSP tree archive design heavily depends on its integration with the search method. For example, for random search, the expected maximum number of leaf nodes is half of the search space, as there is no correlation in the search positions. Such a large archive is obviously useless. For this design to be useful, it has to be integrated with a search method that exploits correlations in the past visits, such that the pruning of the archive occurs frequently and the archive size is kept small throughout the whole process. In this paper, we integrate the tree archive with the GA. We conjecture that the archive can be kept small using the GA. The motivation of the conjecture is based on the Building Block Hypothesis of the GA [4, 17]. The hypothesis suggests that short, low order, schemas that has above average fitness will reproduce better than other schema. Thus at any one time, the GA's population will be populated by several such schemas. Elements within the schemas will be visited more often and they will form clusters. At any one time, the GA will conduct several parallel searches within these clusters, and new potentially good clusters are generated by crossover. The binary space partitioning tries to capture such good schema. Conceptually, the pruning afforded by a particular subspace attempts to reflect the goodness of a particular schema.

III. EXPERIMENTAL RESULTS

In this experiment, a function set $\{f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_6(\mathbf{x})\}$ consisting of six functions are employed to illustrate the performance of the proposed non-revisiting scheme. The six test functions are:

Function 1: Spherical Model

$$f_1(\mathbf{x}) = \sum_{i=1}^{7} x_i^2$$

Function 2: Generalized Rosenbrock's Function

$$f_2(\mathbf{x}) = \sum_{i=1}^{6} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$$

Function 3: Generalized Rastrigin's function

$$f_3(\mathbf{x}) = \sum_{i=1}^{7} \left(x_i^2 - 10\cos(2\pi x_i) + 10\right)$$

Function 4: Generalized Griewank function

$$f_4(\mathbf{x}) = \frac{1}{4000}\sum_{i=1}^{7} x_i^2 - \prod_{i=1}^{7} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

Function 5: Schwefel's Problem 2.26

$$f_5(\mathbf{x}) = -\sum_{i=1}^{7} x_i \sin\sqrt{|x_i|}$$

Function 6: Linear function

$$f_6(\mathbf{x}) = -\frac{2}{10100}\sum_{i=1}^{100} i x_i$$

$f_1(\mathbf{x})$, $f_2(\mathbf{x})$ and $f_6(\mathbf{x})$ are uni-modal whilst $f_3(\mathbf{x})$ to $f_5(\mathbf{x})$ are multi-modal. $f_1(\mathbf{x})$ to $f_5(\mathbf{x})$ are real functions whilst $f_6(\mathbf{x})$ is a pseudo boolean function. This illustrates that the concept is equally applicable to both real and pseudo boolean functions. All six functions are to be minimized.

TABLE 1
DETAILS OF THE SIX TESTING FUNCTIONS

Function	range of **x**	optimum	optimal fitness
f_1	$[-100,100]^7$	[0, 0, …, 0]	0
f_2	$[-30, 30]^7$	[1, 1, …, 1]	0
f_3	$[-5.12, 5.12]^7$	[0, 0, …, 0]	0
f_4	$[-600, 600]^7$	[0, 0, …, 0]	0
f_5	$[-500, 500]^7$	[420.96, …, 420.96]	-2933
f_6	$[0, 1]^{100}$	[1, 1, …, 1]	-1

Table 1 lists the range of **x**, the optimal **x** and the corresponding optimal fitness of the six test functions. The number of divisions d of each dimension of **x** is chosen as 100. The value of d of f_6 is chosen as 2 as it is a pseudo boolean function. In this experiment, the contribution of the proposed scheme is illustrated by comparing the non-revisiting GA (NGA) with the standard GA (with revisits) (GA). The conventional GA: 1-point crossover, 1-point mutation and elitism selection is employed (though other variations of GA are also permissible). The population sizes are chosen as 30. The performance of the scheme is described by two quantities: 1) accuracy – the search power within a fixed number of generations and 2) probability of success.

A. Accuracy Test

In this test, the best fitness found by the NGA and GA are recorded after 60 generations. Since GAs are stochastic algorithms, the results are extracted from 100 independent runs. Table 1 lists the average optimal fitness of the NGA and GA. The improvements I of NGA on the six functions are also provided for illustrating the enhancement made by the non-revisiting scheme. We denote the improvement I of NGA related to GA as the fraction of the error (difference from the global optimum) of GA to the error of GA, i.e. $I = (f_r - f_o) / (f_n - f_o)$, where f_n is the best fitness found by the NGA, f_r is the best fitness found by the GA and f_o is the global optimal fitness. An improvement is made by the NGA if the corresponding I is larger than one.

TABLE 2
EXPERIMENTAL RESULTS OF THE ACCURACY TEST

Function	NGA	GA	I
f_1	4.67	49.62	10.63
f_2	3958.8	141132.7	35.71
f_3	3.2154	8.908	2.77
f_4	0.3914	3.4209	8.77
f_5	-2736.3	-2596.7	1.71
f_6	-0.9804	-0.753	12.65

Seen from Table 2, the enhancement made by the non-revisiting scheme is clear: the best fitness of the six test functions searched by NGA are significantly smaller than those by the GA. One explanation is as follows: When the GA comes into the basin of attraction of a local or global optimum, the chance of generating a revisiting offspring is higher. The random crossover and mutation then constitute a random, revisiting search within the basin, which is much less efficient than a non-revisiting search within the basin. For a global optimum, it facilitates the location of the optimum more quickly. For a local optimum, it facilitates the complete search of the basin, so that the NGA may escape out of the basin sooner.

We define t_n and t_r as the processing time of NGA and GA respectively. The corresponding overhead CPU effort of accessing the BSP tree of NGA is then computed as $O = t_n - t_r$. The mean and standard deviation of t_r for all simulations are around 2 sec. and 0.001 sec. respectively. The average values of O of the first five and the last test function(s) are/is around 0.071 sec. and 0.21 sec. with standard deviations 0.03 sec. and 0.009 sec. respectively. The speed of the CPU used in the simulations is 3.0GHz. C language is used for the simulations.

In real function optimization, the value of d plays an important role as it controls the size of the search space and hence the accuracy. The BSP tree node size S and the node to search space ratio R of the five real functions are illustrated in Fig. 1 to Fig. 5, in order to empirically investigate the computation load of the non-revisiting scheme. In this investigation, the values of d are varied

from 10 to 100. The analysis for f_6 is not done since d is fixed as 2 (pseudo boolean function).

The results of S and R indicate that the archive size of the BSP tree is small even for large search space, i.e. fine resolution solution. The best, average and worst values of S and R, represented as dashed, solid and dotted lines, of the five functions, shown in Fig. 1 to Fig. 5, are within reasonably small ranges, which empirically verify the practicability of the proposed scheme. The archive size ratio R ranges from 10^{-4} (for coarse resolution) to 10^{-10} (for finest resolution). For coarse resolution, the archive affords a quicker and better search and the archive size is not an issue. For the finest resolution, the archive size ratio is still small. In summary, it can be observed that all of them have modest sizes and our implementation of the NGA is an existence proof for its feasibility.

In larger scale optimization, the number of possible solutions non-linearly increases as the resolution increases. As expected, the chance of generating a revisiting offspring is smaller and the size of the BSP tree increases. However, as the individuals of GA are non-uniformly searching the global optimum in the search space, i.e. higher density at the basins of attraction of the local or global optima, the size of the BSP tree is sub-linearly (better than linear) proportional to d (hence the corresponding R is inversely proportional to d). The experimental results suggest that the archive size scales well with the resolution of the search, which is a nice property.

B. Probability of Success (PoS) Test

In this test, we study the enhancement on the probability of success PoS from the proposed non-revisiting scheme. A run is said to be a success if the GA meets a target fitness F_g within 500 generations. The F_g of the six test functions are defined as the corresponding best fitness found in the accuracy test. Table 3 lists the values of F_g of the six test functions.

As in above, 100 independent trials are performed to obtain the averaged PoSs. Table 4 lists the PoSs of NGA and GA. For all the test functions, the PoSs of NGA are superior to that of GA. In summary, a significant improvement on PoS is made by the NGA.

TABLE 3
THE VALUES OF F_g

f_1	f_2	f_3	f_4	f_5	f_6
0	882.2	0.005	0.02	-2855	-0.998

TABLE 4
EXPERIMENTAL RESULTS OF THE POS TEST

Function	NGA	GA
f_1	67%	8%
f_2	49%	2%
f_3	18%	0%
f_4	73%	6%
f_5	24%	3%
f_6	43%	1%

IV. CONCLUSIONS

A fundamental problem in computer science is that stochastic algorithms are re-visiting. It is frequently impossible to record all places that have been visited before. The revisiting problem means that functions are evaluated at the same place more than once. This is clearly a waste of computational resources.

A key contribution of this paper is pointing out that for some classes of stochastic algorithms, it is possible to eliminate *all* such revisits. In particular, it is possible to do it with genetic algorithm (GA) via an archive in the form of a novel dynamic binary space partitioning (BSP) tree. The archive is a random tree that is built up during the search process. Since all "intelligent" search is not proceeding at random, but exploits correlations in the search process, whole visited subtrees can be pruned dynamically during the search. Our critical insight is that the worst case size of such an archive is small for "intelligent" search algorithms such as the GA. So it is a practical solution that solves the problem of revisits in GA conclusively. We propose that it is in part due to the working principle of the GA, which is based on the building block hypothesis. As a significant byproduct, we discover that the tree pruning is isomorphic to an interesting self adaptation mutation operator that is parameter-less. Another key contribution of this paper is that by eliminating revisits, it provides a fresh solution to the problem of premature convergence and the maintenance of diversity. These problems are resolved and eliminated. The method can be applied to both continuous and discrete space problems, i.e., real coded and pseudo boolean functions. The archive design also has the nice property of scaling well with the resolution or problem dimension.

The experimental results are very encouraging. They show that the Non-revisiting GA is clearly superior to its conventional counterparts, and equally importantly, the archive size can indeed be kept small and the overhead of accessing the archive is also small, which means that the design of the archive is practical.

There is a need to extensively test our ideas in this paper in more functions, particularly on real problems. It also seems promising to apply the archive design to other well known stochastic search algorithms, as exploiting correlation is a prevalent theme in general algorithm designs.

ACKNOWLEDGMENT

The work described in this article was supported by a grant from CityU (7001859).

REFERENCES

[1] F. Glover, "Tabu search: 1.," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190-206, 1989.

[2] R.C. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," *Proc. 6th Int. Symp. Micromachine Human Sci.*, vol. 1, pp. 39-43, 1995.

[3] D.H. Wolpert and W.G. Macready, "No free lunch theorems for optimization," *IEEE Trans. on Evolutionary Computation*, vol. 1, no. 1, pp. 67-82, 1997.

[4] J.H. Holland, *Adaptation in natural and artificial systems,* University of Michigan Press, Ann Arbor, 1975.

[5] A.E. Eiben and J.E. Smith, *Introduction to Evolutionary Computing*, Springer 2003.

[6] M.L. Mauldin, "Maintaining diversity in genetic search," *Proc. National Conference on Artificial Intelligence*, pp. 247-250, 1984.

[7] L. Davis, *Handbook of genetic algorithms*, Van Nostrand Reinhold, New York, 1991.

[8] S. Ronald, "Duplicate genotypes in a Genetic algorithm," *Proc. IEEE Int. Conf. on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, pp. 793-98, 1998.

[9] D.E. Goldberg and J. Richardson, "Genetic algorithms with sharing for multimodal function optimization," *Proc. 2nd Int. Conf. on Genetic Algorithms*, Lawrence Erlbaum, pp. 41-49, 1987.

[10] D. Whitley, "The GENITOR algorithm and selection pressure: why rank-based allocation of reproduction trials is best," *Proc. 3rd Int. Conf. on Genetic Algorithms*, Morgan Kaufmann, pp. 116-121, 1989.

[11] L.J. Eshelman, "The CHC adaptive search algorithm: how to have safe search when engaging in nontraditional genetic recombination," *Foundation of Genetic Algorithms*, G.J. E. Rawlins, Ed., California: Morgan Kaufmann, pp. 265-283, 1991.

[12] S.W. Mahfoud, "Crowding and preselection revisited," *Parallel Problem Solving from Nature* 2, R Manner and B. Manderick, Eds., Amsterdam: North-Holland, pp. 27-36, 1992.

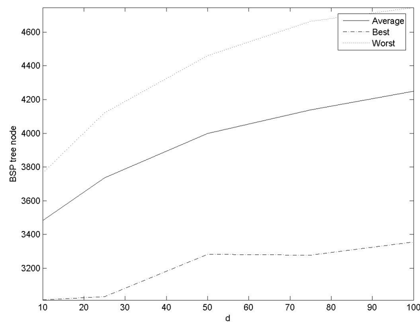
[13] T. Bäck, "The interaction of mutation rate, selection and self-adaptation within a genetic algorithm", *Parallel Problem Solving from Nature* 2, R Manner and B. Manderick, Eds., Amsterdam: North-Holland, pp. 85-94, 1992.

[14] T.E. Davis and J.C. Principe, "A Markov chain framework for the simple genetic algorithm," *Evolutionary Computation*, vol. 1(3), pp. 269-288, 1993.

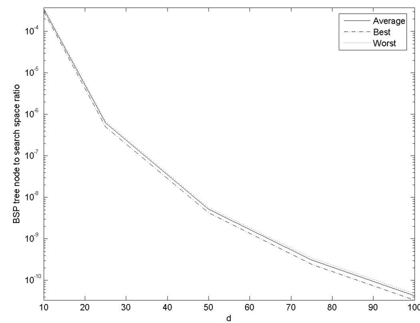
[15] A.E. Eiben, R. Hinterding, and Z. Michalewicz, "Parameter control in evolutionary algorithms", *IEEE Transactions on Evolutionary Computation*, vol. 3(2), pp. 124-141, 1999.

[16] J. Knowles and D. Corne, "Properties of an adaptive archiving algorithm for storing nondominated vectors," *IEEE Trans. on Evolutionary Computation*, Vol. 7, no. 2, pp. 100-116, 2003.

[17] S. Forrest, M. Mitchell, "Relative building-block fitness and the building-block hypothesis," *Proc. 2nd Workshop on the Foundations of Genetic Algorithms (FOGA)*, D. Whitley, Ed., Morgan Kaufmann, San mateo, CA, pp. 109-126, 1993.

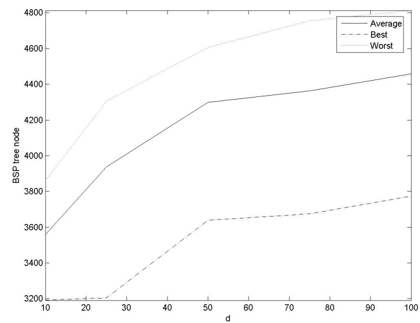


(a)

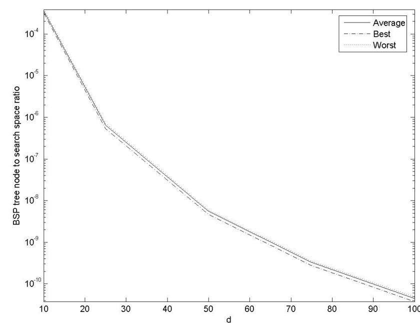


(b)

Fig. 1. Results about the spherical function: (a) BSP tree node and (b) the corresponding *R* plotted against the number of divisions.

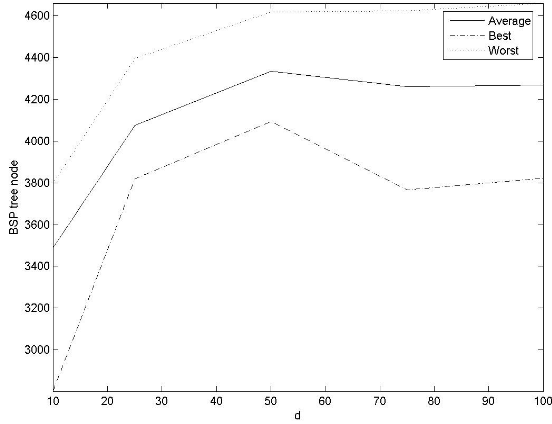


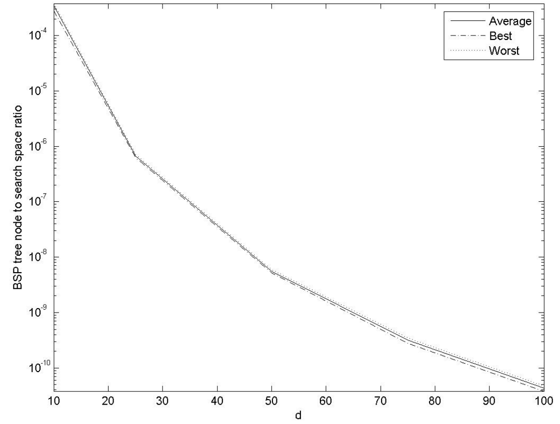
(a)



(b)

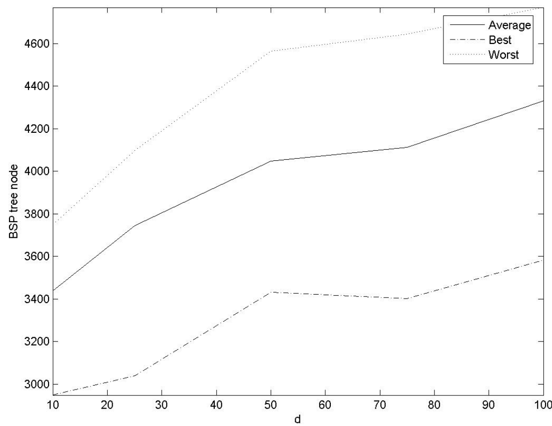
Fig. 2. Results about the generalized Rosenbrock's function: (a) BSP tree node and (b) the corresponding *R* plotted against the number of divisions.

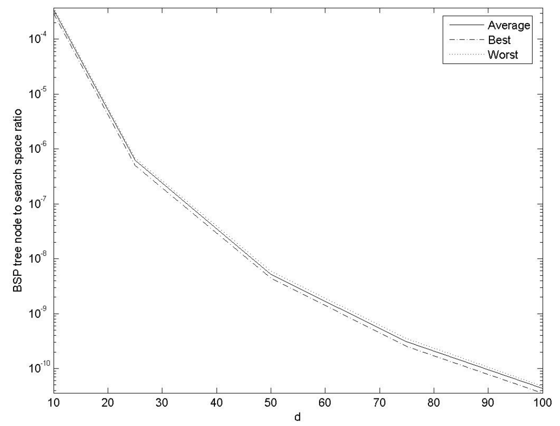




(a) (b)

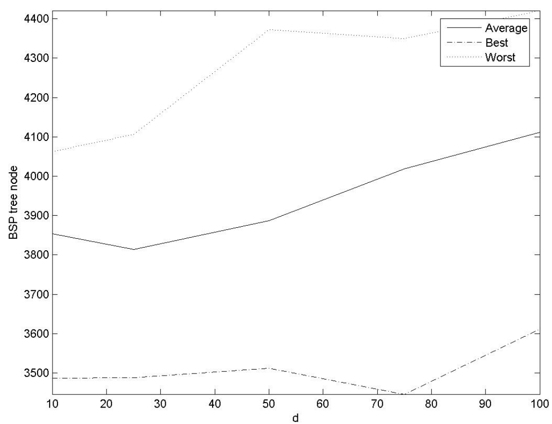
Fig. 3. Results about the generalized Rastrigin's function: (a) BSP tree node and (b) the corresponding *R* plotted against the number of divisions.

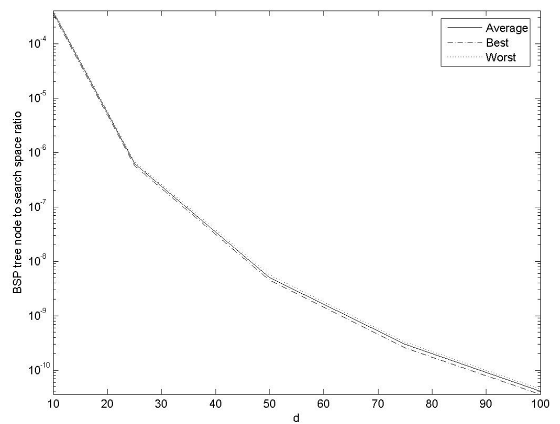




(a) (b)

Fig. 4. Results about the generalized Griewank's function: (a) BSP tree node and (b) the corresponding *R* plotted against the number of divisions.





(a) (b)

Fig. 5. Results about the Schwefel's problem 2.26: (a) BSP tree node and (b) the corresponding *R* plotted against the number of divisions.