Applying Non-Revisiting Genetic Algorithm to Traveling Salesman Problem

Shiu Yin Yuen and Chi Kin Chow

Abstract— **In [1], we propose non-revisiting genetic algorithm (NrGA) and apply it to a set of bench mark real valued test functions. NrGA has the advantage that it is non-revisiting, i.e. a visited point will not be visited again. This provides an automatic mechanism for diversity maintenance which does not suffer from premature convergence. Another advantage is that it supports a parameter-less adaptive mutation mechanism. In this paper, we show how NrGA can be adapted to a real world combinatorial optimization problem – the famous traveling salesman problem (TSP). Comparison with genetic algorithm (GA) (with revisits and standard mutation) is made. It is shown that NrGA gives superior performance compared to GA. Moreover, it gives the same stable performance using different types of mutation operators. Moreover, turning off GA's mutation operator but only use the NrGA inherent parameter-less adaptive mutation gives the best performance.**

I. Introduction

Many stochastic optimization algorithms do not memorize places that they have visited. Perhaps the most famous exception is Tabu search [2], which explicitly records recently visited solutions in a Tabu list. These solutions are precluded from revisits unless an aspiration criterion is satisfied. However, not all revisits are recorded as solutions are discarded when the Tabu list is full.

Theoretically, by 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 the former has the same performance but with fewer function evaluations than the latter. Thus it is always beneficial to eliminate all the revisits.

Genetic algorithm (GA) is a famous stochastic optimization algorithm. It mimics the evolutionary process of a population of individuals over time. The hall mark of GA is the provision of a population (multiple parallel search capability), selection (survival of the fittest), crossover (sexual reproduction) and mutation (random incremental changes). GA is closely related to Evolutionary Strategies (ES) and Evolutionary Programming (EP). In GA, crossover is a key operator whilst mutation is usually a background operator. Traditional GA uses binary coded strings called chromosomes, but gray code, integer code and real code are also a common coding. Though mutation is usually a background operator, many authors suggest that an adaptive mutation schedule would enhance GA performance significantly, see for example survey [4] and book [5]. In ES and EP, self adaptive mutation is the key operator, and traditional ES and EP are real coded [6]. However, the binary coded or variant form of GA allows it to be applied to both discrete/combinatorial or continuous/real problems. The schools of GA, ES and EP are best viewed as different dialects of the same fundamental evolutionary algorithm (EA) [6].

Shiu Yin Yuen and Chi Kin Chow are with the Department of Electronic Engineering, City University of Hong Kong, Hong Kong SAR, China (e-mail: {kelviny, chowchi}@cityu.edu.hk).

It is confirmed by numerous experiments that *diversity maintenance* through *duplicate removal* can enhance the performance of GA significantly. Mauldin [7]'s uniqueness operator only allows 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 [8] reports that a binary coded GA which removes duplicates in the population results in superior performance in a comparable number of child evaluations. Recently, Friedrich et al. [9] show by expected time complexity analysis that for an EA with a population greater than 1, a uniform mutation and with no crossover, the use of the above remove duplicate method changes the time complexity of optimizing a plateau function from exponential to polynomial. Crossover is omitted in the above work because it is a difficult operator to analyze. This result provides some theoretical support to the power of duplicate removal. While the above researches compare each child with each solution in the current population, for a population with δ individuals, δ comparisons need to be made. To further enhance the efficiency, Ronald [10] reports the use of hash table to reduce the number of comparisons to *O(1)*. However, these efforts only compare a child with the current population. It does not guarantee no revisits - i.e. no duplicates in the entire search. Povinelli and Feng [11] use a small hash table to store all visited individuals. When the table is full, it is discarded and a larger table is used. Kratica [12] uses a small fixed size cache to store all visited individuals. When the cache is full, an old entry is discarded to make place for a new entry using the least-recently-used strategy. He investigates experimentally the best cache size for a plant location problem. These two results extend the use of memory to beyond one generation. They report a substantial improvement to GA by adding the hash table or cache, but they do not store all the individuals and thus do not guarantee no revisits.

From another angle, *premature convergence* is an undesirable phenomenon often reported in the literature as to cause poor performance of GA. It refers to the state when all of the population consists of a single type of individual that is sub-optimal. It is generally agreed by the GA community that to prevent premature convergence, an appropriate diversity in

978-1-4244-1823-7/08/$25.00©2008 IEEE

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 GA reduces to parallel mutation climbing. Numerous operators that modify the selection, crossover or mutation to diversify the population have been proposed. Two well known examples are fitness sharing [13] and island model [14].

From yet another angle, it is known from studies [4, 5] that an adaptive mutation rate is beneficial for the GA/ES/EP to find the global optimum more efficiently. Three types of parameter controls are distinguished in [4]: A time evolution equation is used in deterministic control. Feedback from the search process is used in adaptive control. The parameters themselves are coded as part of the solutions and evolve in self adaptive control. However, all inevitably involve introducing more control parameters and the determination of the suitable values requires parameter tuning either at the GA level or meta-GA level. The tuning is itself a difficult problem [5].

In [1], we proposed a novel search method known as non-revisiting genetic algorithm (NrGA), which guarantees no revisits during the GA execution. It has four key strengths: (1) The NrGA takes advantage of the "free lunch" result that a non-revisiting algorithm is superior to a revisiting algorithm. (2) It automatically assures of diversity maintenance through duplicate removal. (3) It has by nature *no* premature convergence problem. (4) It naturally deduces an adaptive mutation operator that is parameter-less.

The basic idea is summarized below. For full details of the algorithm and pseudo code, please refer to [1].

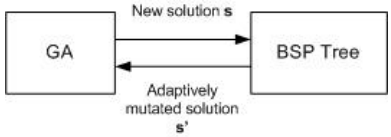


Fig. 1 Communication between GA and the BSP tree

NrGA may be conceptualized as a GA that interacts with a *binary space partitioning* (BSP) tree archive (fig. 1). A GA can be visualized as generating a sequence of solutions *sq = (s(1), s(2), ...)*. The function of the BSP tree is to serve as an archive *Ar* that stores the visited solutions. If the solution *s(t)* has been visited before, it returns an *adaptively mutated* solution *s'* that is unvisited.

BSP has been used in computer graphics and computational geometry as a more general and efficient method than Oct-tree and K-d tree [15],[21]. Here it is used as an efficient data structure to enable fast query of whether a solution has been visited. The results of GA are used to gradually build up a random BSP tree which binary partitions the search space successively.

Initially, the tree has only the root node, which represents the entire search space. Each new solution *s* generated by GA will add one node to the tree that stores the solution *s*. (If *s* is found to be a revisit, an adaptive mutated new solution *s'* will be stored.) The position where the node will be added depends on the existing structure of the tree. For each new solution *s*, a depth first search is initiated on the binary tree. To illustrate the binary partitioning concept, suppose both the left and right nodes exist. Let the left node be the visited solution *s(L)* and the right node be the visited solution *s(R)*. Define the *comparing dimension cd* as the dimension *i* in *s(L)* and *s(R)* that has the largest difference, i.e.

$$cd = \arg\max_i \left| s(L)_i - s(R)_i \right| \quad (1)$$

where $s(L)_i$ is the value at the ith dimension of $s(L)$. In this paper, the Euclidean metric is used. Other metric, such as the Hamming metric, may also be used.

s will go to the left branch if it is closer or equidistant to *s(L)* at the comparing dimension, i.e.

$$\left| s_i - s(L)_i \right| \le \left| s_i - s(R)_i \right| \quad (2)$$

and vice versa. In this way, *s(L)* and *s(R)* binary partitions the search space into two "subspaces" (more precisely, hyper-rectangular boxes).

In general, each node of the binary tree has at most two child nodes. If the left, right, or both nodes are absent, a branching occurs to create one such node that stores the solution *s*.

Along the search, revisit is checked. *s* is a revisit if a node *s()* is found such that $|s - s()| = 0$. This check is called a *revisit check*. Assuming the tree is balanced, only *O(logN)* comparisons is needed to determine whether a revisit has occurred, where *N* is the number of solutions generated by GA so far. Conceptually, it partitions the search space based on the probability distribution of solutions generated by GA.

It is easy to check the size of the subspace *R* under each node. This allows for either *branching* or *pruning*. These two mechanisms provide for the parameter-less adaptive mutation.

The mechanism is as follows: Nodes are marked either *open* or *closed*. Initially, all the nodes are open. When a revisit occurs, an open node *s()* is encountered that is the same as the solution *s*. Two actions may occur: 1) (Branching): If the subspace *R* under the node has not all been visited, a branching occurs to create an unvisited solution $s' \in R$ randomly chosen from *R*. 2) (Pruning): If *R* has all been revisited, the node is marked closed. This initiates a depth first order backtracking from the node to find *the nearest neighbor region* of *R*: *R'*. The unvisited solution $s' \in R'$ is randomly chosen from *R'*.(Note that this operation is slightly different from that presented in [1], which finds the nearest neighbor in *R'*). If more than one nearest neighbor exists, one is chosen randomly. Along the way, if it is found that whole subspaces have been visited, the sub-tree is pruned and the nodes of the sub-tree is marked closed.

The adaptive mutation operator will adjust its mutation step size automatically according to its position in the tree. Interestingly, the more visited is a branch, the smaller will be the step size and vice versa. This is reasonable because the more visited is the branch, the more the GA – itself a complex heuristic stochastic process - reasons as promising to exploit the underlying subspace, hence the smaller mutation and vice versa. Moreover, the operator will 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 at least the nearest unvisited neighbor. These two mechanisms naturally constitute a parameter-less adaptive mutation operator.

Note that the algorithm automatically guarantees that each individual in a current population is different from each other without the need to introduce any special operator. Thus it does not suffer from the premature convergence problem.

In [1], we report the experimental comparison with a real coded GA (with revisit and standard mutation) on 6 popular bench mark test functions. NrGA outperforms GA significantly in both a) the quality of the solutions found using the same number of fitness evaluations; and b) the probability of success of achieving a target fitness, allowing a larger maximum number of fitness evaluations.

So far NrGA has been tested in simulated bench mark functions only. It would be interesting to apply NrGA to a real world problem to understand its practical merits and performance. In this paper, we apply NrGA to a real world combinatorial optimization problem – the traveling salesman problem (TSP). The paper is organized as follows. Section 2 introduces the design of NrGA tailored made for TSP. Section 3 reports the experimental results. Section 4 gives the conclusion.

II. NrGA Design for TSP

TSP is a famous NP hard problem. GA has been applied to TSP. The best performing GA to date for TSP uses GA hybridized with a local search [16]. In this paper, we use GA or NrGA alone without the local search as we wish to investigate the relative performance of GA and NrGA. In this section, we propose a method to apply NrGA to TSP.

Definition: A valid tour of an *n*-city TSP instance

A tour $\mathbf{T} = [t_1,\ldots, t_n]$ of an *n*-city TSP instance is said to be a valid tour if 1) $t_i \in [1, n]$ and 2) $t_i \neq t_j$ for all $i \neq j$. €

i denotes the i[th] *stop* in the tour and t_i is the *city number* of the i[th] city visited. The problem is to find a valid tour T that has the minimum distance. The search space is of size $n!$. A full search quickly becomes impossible when *n* grows large.

For GA and NrGA, a chromosome is represented as a valid tour T, i.e. integer coding is used. The GAs designed for TSP (see survey in [16] for example) make sure that the initial population will only generate valid tours. Moreover, crossover and mutation operators are specifically designed by researchers [6, 18-20] so that a valid tour always returns a valid tour.

When applying NrGA to real valued function optimization [1], the size of the subspace under a node is $R = \Pi[L_i, U_i] \subset \Re^n$. L_i and U_i are the lower and upper bounds of the values at the i^{th} dimension. A closed node indicates that the subtree under it has been visited. For a visit to an open node, iff $\#(R) = 1$, then the node constitutes a single point in the search space. It is marked closed and a backtracking is initiated.

In the context of TSP, L_i and U_i are the lower and upper bounds of *city number* at the i[th] *stop*. By nature of the binary space partitioning, *R* will contain at least one valid tour. However, in TSP, even if $\#(R) > 1$, it may still mean that *there is only a single valid tour*. Fortunately, we only need to devise a scheme to make the binary decision of whether *R* represents either (a) a unique valid tour or (b) more than one valid tour. The former will initiate the backtracking and/or pruning whereas the latter will initiate branching to create a new node.

A simple *tour uniqueness verification* algorithm is introduced for the above function:

Algorithm A1 (tour uniqueness verification):

Input: 1) $R = \Pi[L_i, U_i] \subset \Re^n$

1. Define an $n \times n$ matrix $\mathbf{M} = \{m_{i,j}\}$ of binary elements where
$$m_{i,j} = \begin{cases} 1 & \text{if } j \in [L_i, U_i] \\ 0 & \text{otherwise} \end{cases} \quad (3)$$
2. for $i = 1$ to n
   if $\sum_{j=1}^{n} m_{ij} = 1$
      find k for which $m_{i,k} = 1$
      call *(i,k)* a *1-1 mapping*
      assign $m_{j,k} = 0$ for all $j \neq k$
   end if
   next *i*
3. repeat step 2 until in the current cycle, no new 1-1 mapping is discovered
4. if all stop-city are 1-1 map, output *0*, otherwise output *1*.

Output: 1) *0* – a unique valid tour; 2) *1* – more than one valid tour

The idea is to check whether a stop *i* can be found that must be mapped to a city *k*. If so, city *k* cannot map to stops other than *i*. The check is then repeated. If there is a unique tour, an assignment should eventually be found such that all stops are 1-1 mapped to cities.

Lemma: If only a unique valid tour exists, algorithm A1 will always output 0 - a unique valid tour.

Proof:

Given a unique valid tour **T**, rename city number such that city number is identical to stop number. Suppose $\mathbf{M_1}$ is the output of algorithm A1, remove all rows and columns that have 1-1 mappings to produce submatrix $\mathbf{M_2}$. Since binary space partitioning will always maintain one and only one block of '1' in each row, $m_{i(i-1)}$ or $m_{i(i+1)}$. Since $m_{ij} \neq m_{ji}$ for all $i \neq j$, otherwise exchange *i* and *j* will give another valid tour, the '1' entry must be $m_{i(i+1)}$. But this cannot be satisfied for $i=n$. Therefore given a unique valid tour **T**, it cannot happen that in its reduced form, $\mathbf{M_2}$ contains more than one '1' in each row, no matter how small is $\mathbf{M_2}$. The lemma is proved by contradiction. □

Corollary: Given more than one valid tour, some *i* and *j*, $i \neq j$,

must exist such that $m_{ii}=m_{jj}=m_{ij}=m_{ji}=1$. We can exchange i and j.

If a solution **x** is a revisit, it has to mutate to a non-revisited position. According to [1], the mutated **x**: **y**, is randomly selected from the range R where **x** belongs to. Unlike the real function optimization, some points in the search space of TSP are invalid tours. Thus, a TSP specific adaptive mutation is introduced.

Algorithm A2 (adaptive mutation):

Input: 1) $R = \Pi[L_i, U_i] \subset \Re^n$ and 2) a revisit $\mathbf{x} \in R$

1. Compute the reduced **M** of R by steps 1 - 3 of algorithm *A1*.
2. Randomly select a pair of stops (a and b) subject to $a \neq b$ and $m_{a,x_b} = m_{b,x_a} = 1$
3. Generate a valid tour $\mathbf{y} = [y_1, y_2, \ldots, y_n]$ where

$$y_k = \begin{cases} x_b & \text{if } k = a \\ x_a & \text{if } k = b \\ x_k & \text{otherwise} \end{cases} \quad (3)$$

Output: the mutated **x** = **y**

The corollary above assures that one can always find feasible mutation.

Example:

The TSP instance has three cities. The search space X is $[1,3]^3$ which has 3! = 6 possibilities. Suppose GA randomly generates *sq = (S(1), S(2), S(3), S(4))* = ($\mathbf{s_1}$ = [1,2,3], $\mathbf{s_2}$ = [3,1,2], $\mathbf{s_3}$ = [3,2,1], $\mathbf{s_4}$ = [3,2,1]). Note that $\mathbf{s_4}$ is a revisit.

Initially, the archive is empty. It consists only of the root node (fig. 2a). $\mathbf{s_1}$ is inserted as an open child node of the *root* (fig. 2b). $\mathbf{s_2}$ is next generated by GA. Since there is only one node, the search visits node $\mathbf{s_1}$. A revisit check of $\mathbf{s_2}$ with $\mathbf{s_1}$ is conducted. Since $|\mathbf{s_2} - \mathbf{s_1}| > 0$, $\mathbf{s_2}$ is not a revisit. Hence it is inserted as another open child of the *root* (fig. 2c). The comparing dimension *cd* is *1* because |3-1| > |1-2| = |2-3|. As a result, this insertion is equivalent to partitioning X into two sub-spaces: $X_1 = [1,2]\times[1,3]^2$ and $X_2 = [3]\times[1,3]^2$ where $\mathbf{s_1} \in X_1$ and $\mathbf{s_2} \in X_2$. A tour uniqueness verification is done for X_1 and X_2: For X_1, the corresponding $\mathbf{M_1}=\begin{bmatrix}1&1&0\\1&1&1\\1&1&1\end{bmatrix}$ has more than one valid tour. For X_2: $\mathbf{M_2} = \begin{bmatrix}0&0&1\\1&1&1\\1&1&1\end{bmatrix}$. Clearly, the first stop can only be city 3. Hence $\mathbf{M_2}$ is reduced to $\begin{bmatrix}0&0&1\\1&1&0\\1&1&0\end{bmatrix}$. No more reduction can be made. Hence $\mathbf{M_2}$ also has more than one valid tour. $\mathbf{s_3}$ is next generated by GA. Since $|s_{3,1} - s_{2,1}| \leq |s_{3,1} - s_{1,1}|$, it goes down the right branch. A revisit check of $\mathbf{s_3}$ with $\mathbf{s_2}$ is conducted. Since $|\mathbf{s_3} - \mathbf{s_2}| > 0$, $\mathbf{s_3}$ is not a revisit and it is inserted as an open node under $\mathbf{s_2}$ (fig. 2d). After this insertion, X_2 is further divided into two non-overlapping sub-spaces: [3,3] × [1,1] × [2,2] and [3,3] × [2,2] × [1,1]. According to the tour uniqueness verification, each of these sub-spaces has one valid tour, in which all possible valid tours in X_2 are visited. Thus, $\mathbf{s_3}$ is pruned and $\mathbf{s_2}$ is marked as a closed node (gray -filled circle in fig. 2e). $\mathbf{s_4}$ is next generated by GA. By similar reasoning, it would select the right branch to $\mathbf{s_2}$. Since $\mathbf{s_2}$ is closed, $\mathbf{s_4}$ is a revisit and an adaptive mutation should be performed. The mutation begins by backtracking. According to *Case* 2 of step 4 of the algorithm *A1* in [1], the mutated $\mathbf{s_4}$: $\mathbf{s_4}$**'** should be within X_1 and s_4' is generated by the algorithm *A2* for $R = X_1$ and $\mathbf{x} = \mathbf{s_1}$. In the adaptive mutation, the reduced **M** of X_1 is $\begin{bmatrix}1&1&0\\1&1&1\\1&1&1\end{bmatrix}$. The stop pair is randomly chosen as 1 and 2. Thus, $\mathbf{s_4}$**'** is computed as [2,1,3] and added under $\mathbf{s_1}$ (fig. 2f).

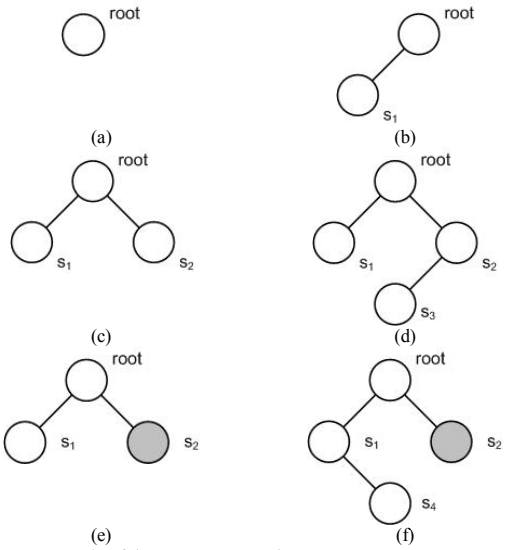


Fig. 2 An example of the BSP tree operations.

III. Experimental Results

A. Experiment Design

In this section, five symmetric TSP instances containing 24, 29, 48, 70 and 130 cities are employed to illustrate the performance of the NrGA on solving TSPs. All TSP instances in this section are taken from the TSPLIB [17]:

TSP instance 1: Groetschel 24-city problem (T_1*: Gr24*)
TSP instance 2: Bavaria 29-city problem in geographical distances (T_2*:* Bayg29)
TSP instance 3: Groetschel 48-city problem (T_3*:* Gr48)
TSP instance 4: Smith and Thompson 70-city problem (T_4*:* ST70)
TSP instance 5: Churritz 130-city problem (T_5*:* Ch130)

Table I lists the actual optimal distances D_o of these five TSP instances $\{T_i\}$.

TABLE I
OPTIMAL DISTANCES OF THE TSP INSTANCES

T_i	Gr24	Bayg29	Gr48	ST70	Ch130
D_o	1272	1610	5046	675	6110

The accuracy of GA is represented by the corresponding optimal fitness found in a fixed number of generations. The accuracy of NrGA is compared with GA. The performance of NrGA and GA are concluded by 100 independent runs. The experiments are repeated with different combinations of crossover and mutation operators commonly used in GA for TSP [6, 18-20]: The *Order Based Crossover* (XO) operator is employed. Since the effect of mutation is to be studied, four TSP specific mutation operators are employed. They are 1) *Exchanging Neighbor Mutation* (EN); 2) *Exchange Mutation* (EX); 3) *Simple Inversion Mutation* (SI); 4) *Displacement Mutation* (DP).

Note that the above mutation operators are applied to GA. The NrGA constitutes its separate parameter-less adaptive mutation operator (AM). Thus, for each TSP instance, NrGA is performed five times:

1) NrGA with EN and AM (EN + AM)
2) NrGA with EX and AM (EX+AM)
3) NrGA with SI and AM (SI+AM)
4) NrGA with DP and AM (DP + AM)
5) NrGA with AM only (AM)

On the other hand, GA is repeated four times for each TSP instance:

1) GA with EN (EN)
2) GA with EX (EX)
3) GA with SI (SI)
4) GA with DP (DP)

The population size is chosen to be 200. The crossover probability is 1.0. (200+200) selection is used. Since the last two TSP instances have more cities, the NrGA is run for 300 generations for the first three TSP instances and 1000 generations for the last two TSP instances. Since NrGA involves the management of the BSP tree, it invokes an overhead. It would be unfair to compare GA and NrGA over a fixed number of generations. For a fair comparison, the amount of time needed to run NrGA is recorded. GA is allowed to run the same amount of time.

B. Qualitative Characteristics

Table II lists the best fitness found by NrGA and GA. The bolded value indicates that the corresponding mutation operator outperforms the others in a particular TSP instances. The odd rows represent the results of NrGA and the even rows represent the results of GA. Seen from the table, AM is superior to other eight mutation operators in the five TSP instances. Meanwhile, for each mutation operator, the result of NrGA is better than that of GA. For example, considering the EN operator at *Gr24*, the optimal fitness found by NrGA (EN+AM) is 1325.74 whilst the optimal fitness found by GA (EN) is 1948.22. For the SI operator at ST70, the optimal fitness found by NrGA (SI+AM) is 1025.24 whilst the optimal fitness found by GA (SI) is 1882.04. In addition, it is observed that the results at the odd rows (found by NrGA) are consistent. For example, the range of optimal fitness of *Gr24* is from 1289.77 to 1337.02. On the other hand, the range of optimal fitness of Gr24 found by GA is from 1378.06 to 1948.22. Figures 6 - 10 (in coloured curves) show the convergence curves of NrGA and GA on solving the *Gr24*, *Bayg29*, *Gr48*, *ST70* and *Ch130* TSP instances respectively. NrGA outperforms GA for all discussed operators. The most significant accuracy improvement of NrGA over GA can be observed from the cases of AM. Furthermore, the results infer that choice of the combination of genetic operators affects to a large extent the performance of the GA. The differences of the best distances found with different operator combination can be very large, i.e. 200 for *Bayg29* and 6000 for *Gr48*. On the other hand, the influence of the operator combination is minor in NrGA.

Figure 11(a) shows the convergence curves of GA with (EN, EX, SI, DP) and NrGA with AM against iterations for Gr24. Seen from the figure, all curves except AM are premature converging after the 150th iteration while the improvement by AM lasts for 300 iterations. Note that the optimal fitness is 1272 and the AM curve is converging to it.

We define the diversity v of an iteration as the averaged standard deviation of the currently generated offspring:

$$V = \frac{1}{D}\sum_{i=1}^{D}\sigma_i \tag{4}$$

where

$$\sigma_i^2 = \frac{1}{n}\sum_{j=1}^{n}\left(x_{j,i}-\mu_i\right)^2 \text{ and } \mu_i = \frac{1}{n}\sum_{j=1}^{n}x_{j,i} \tag{5}$$

and n is the number of offspring. Figure 11(b) shows the diversities of GA with (EN, EX, SI, DP) and NrGA with AM for Gr24 against iterations. Seen from the figure, AM has the highest diversities. In particular, the diversity of EN is worse. It drops to 0 by the 50th iteration. This shows that the diversity maintenance is best in NrGA. It constantly explores the search space; so a continuous improvement is expected.

TABLE II
THE BEST FITNESS VALUES FOUND BY NRGA AND GA

	Gr24	*Bayg29*	*Gr48*	*ST70*	*Ch130*
EN+AM	1325.74	1718.45	6395.96	977.59	12581.37
EN	1948.22	2688.38	12678.19	2313.10	32525.70
EX+AM	1325.47	1712.08	6577.50	986.92	13280.53
EX	1378.06	1815.29	7256.56	1123.09	14802.98
SI+AM	1337.02	1746.37	6922.64	1025.24	13345.62
SI	1638.88	2253.78	10202.51	1882.04	25696.49
DP+AM	1289.77	1640.98	5529.04	771.87	8765.99
DP	1415.81	1822.35	6124.03	873.88	8979.69
AM	**1276.87**	**1624.57**	**5086.72**	**710.12**	**8064.71**

C. Archive Size

Table III lists the statistics of the archive sizes of NrGA with the five mutation operators. The upper bound of the archive size in this test is $S_{upper} = 301 \times 200 = 60{,}200$ for the first three TSP instances and 200,200 for the last two instances. In our experiment, the S_{worst} of the five TSP instances are 59,545, 59,925, 60,081, 200,008 and 200,093. By comparing using a PC that is commonly configured with

256MB memory, the size of the BSP tree consisting of 200,093 nodes are just 1.87% of the memory.

TABLE III
THE STATISTICS OF THE ARCHIVE SIZES

		Min.	*Max.*	*Mean*	*Stdev.*
Gr24	EN + AM	41897	55315	49033	2949
	EX + AM	55293	59806	58393	906
	SI + AM	50555	59156	56602	1720
	DP + AM	54795	59545	58191	883
	AM	42476	55616	49082	3052
Bayg29	EN + AM	49180	58477	55092	2321
	EX + AM	57705	59908	59492	370
	SI + AM	53278	59606	58671	830
	DP + AM	57925	59925	59365	385
	AM	47259	58267	54347	2592
Gr48	EN + AM	59044	59960	59784	136
	EX + AM	59982	60081	60040	19
	SI + AM	59685	60027	59967	63
	DP + AM	59977	60076	60040	21
	AM	58855	59950	59801	138
ST70	EN + AM	189751	199606	198142	1910
	EX + AM	199410	200008	199875	107
	SI + AM	198694	199879	199590	231
	DP + AM	197607	199964	199653	337
	AM	189113	199609	198204	1833
Ch130	EN + AM	199984	200031	200010	11
	EX + AM	200067	200091	200081	5
	SI + AM	200047	200077	200062	7
	DP + AM	200068	200093	200083	5
	AM	199979	200022	200002	13

IV. CONCLUSIONS

The Non-revisiting genetic algorithm (NrGA) has its root in a corollary of the no free lunch theorems: There is a strictly positive gain of converting any revisiting algorithm to a non-revisiting one. In this paper, we apply the idea to a NP hard problem – the traveling salesman problem (TSP), as it would be interesting to gauge the performance gain of a non-revisiting stratagem on a NP class problem, especially since many NP hard problems of disparate nature is known to be related to each other.

A novel way to adapt NrGA to the TSP setting is reported and a comparison is performed between NrGA and GA for 5 popular bench marks in the TSP library. Since NrGA has overhead in accessing and modifying the binary space partitioning (BSP) archive, the raw computation time is used for a fair comparison.

It is found that 1) firstly, there is a significant improvement in the quality of the tour when NrGA is used; 2) Secondly, no matter what TSP mutation operator is used in GA, after adding the Nr feature, the performance improves and the nearly a constant gain is observed. 3) Thirdly, if one disuses the TSP mutation operator of GA and merely use the NrGA's unique parameter-less adaptive mutation operator, the performance improvement is the largest. This suggests that NrGA adaptive mutation operator is itself a novel good and stable mutation operator for TSP. 4) Finally, for TSP problems, experimental results corroborate that NrGA suffer no premature convergence problem whereas GA suffers from it.

Though some pruning occurs, it is found that the BSP archive still requires a significant amount of memory usage for TSP problems. Thus the limitation of the NrGA is that it is restricted by the memory available – which is the number of fitness evaluations that the GA is allowed. For normal GA setting, this is usually not a problem.

Since our goal is to compare GA with revisits and GA without revisits, we have not incorporated local search in the GA. The current wisdom in TSP research is that GA + local improvement heuristic gives the best performance. It would be interesting to use the memetic (NrGA + local improvement heuristic) to solve large bench mark problems in the future.

ACKNOWLEDGMENT

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

REFERENCES

[1] S.Y. Yuen and C.K. Chow, "A Non-revisiting genetic algorithm," in Proc. of *IEEE CEC Conf.*, pp. 4583 – 4590, 2007.

[2] F. Glover and M. Laguna, *Tabu search*, Kluwer Academic Publishers, 1997.

[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] 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.

[5] F.G. Lobo, C.F. Lima and Z. Michalewicz (Eds.), *Parameter setting in evolutionary algorithms*, Springer, 2007.

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

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

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

[9] T. Friedrich, N. Hebbinghaus and F. Neumann, "Rigorous analyses of simple diversity mechanisms," *Proc. GECCO*, July 2007.

[10] S. Ronald, "Duplicate genotypes in a Genetic algorithm," In *Proc. IEEE Int. Conf. on Evolutionary Computation,* pp. 793-98, 1998.

[11] R. J. Povinelli and X. Feng, "Improving genetic algorithms performance by hashing fitness values," *Proc. Artificial Neural Networks in Engineering*, pp. 399-404, 1999.

[12] J. Kratica, "Improving performances of the genetic algorithm by caching," *Computers and Artificial Intelligence*, vol. 18, no. 3, pp. 271-283, 1999.

[13] 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.

[14] E. Alba and M. Tomassini, "Parallelism and evolutionary algorithms," *IEEE Trans. Evolutionary Computation*, vol. 6, no. 5, pp. 443-462, 2002.

[15] D. Hearn and M.P. Baker, *Computer graphics with OpenGl*, 3rd Ed., 2004.

[16] H.D. Nguyen, I. Yoshihara, K. Yamamori and M. Yasunaga, "Implementation of an Effective Hybrid GA for Large-Scale Traveling Salesman Problems," *IEEE Trans. on Systems, Man and Cybernetics*, part B, vol. 37, no. 1, pp. 92-99, Feb. 2007.

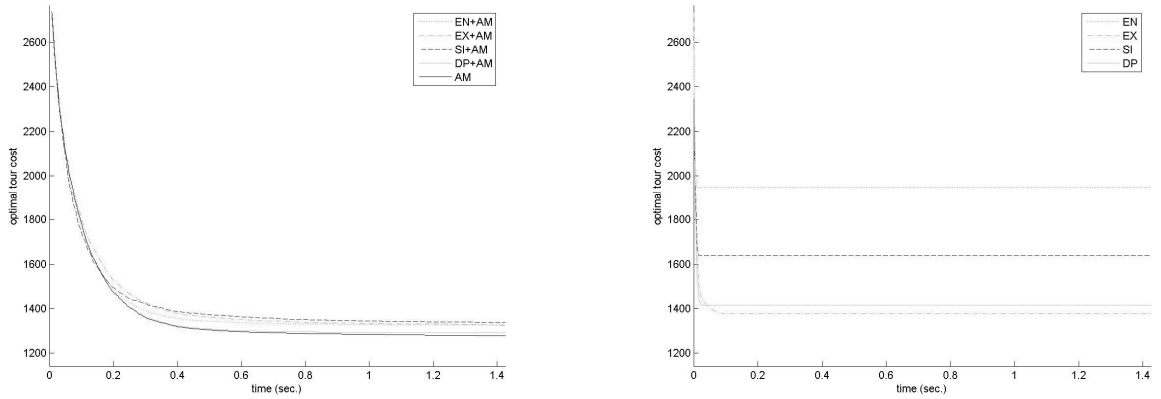
[17] TSPLIB homepage: http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html

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

[19] W. Banzhaf, "The "Molecular" Traveling Salesman," *Biological Cybernetics*, vol. 64, pp. 7 – 147, 1990.

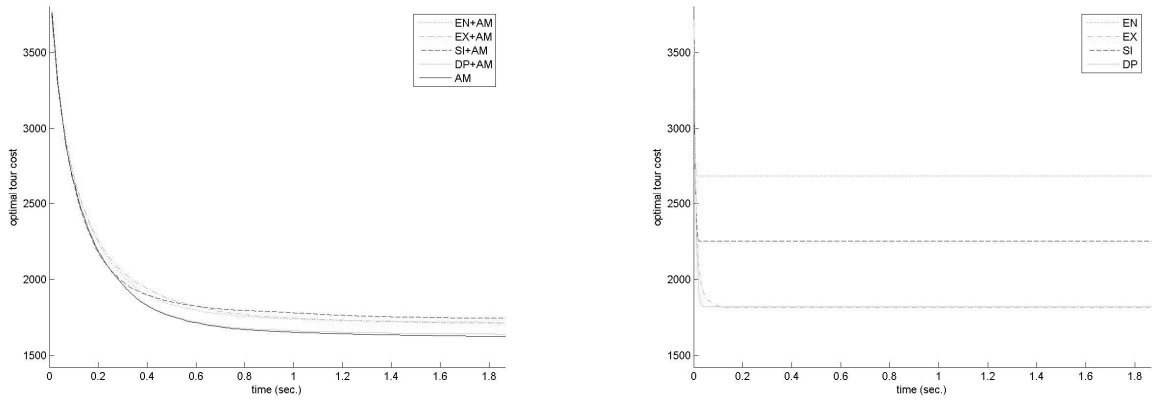
[20] Z. Michalewicz, *Genetic Algorithm + Data Structure = Evolution Programs*. Springer Verlag, Berlin Heidelberg, 1992.

[21] M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, *Computational geometry, algorithms and applications*, 2nd Ed. 2000.



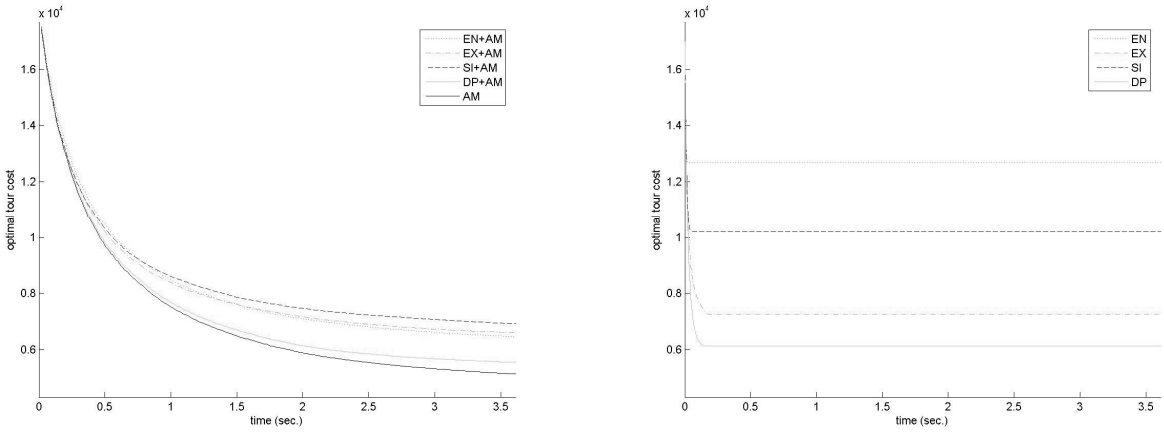
(a) (b)

Figure 6. Averaged convergence curves of Gr24 under different mutation operators: (a) NrGA and (b) standard GA.



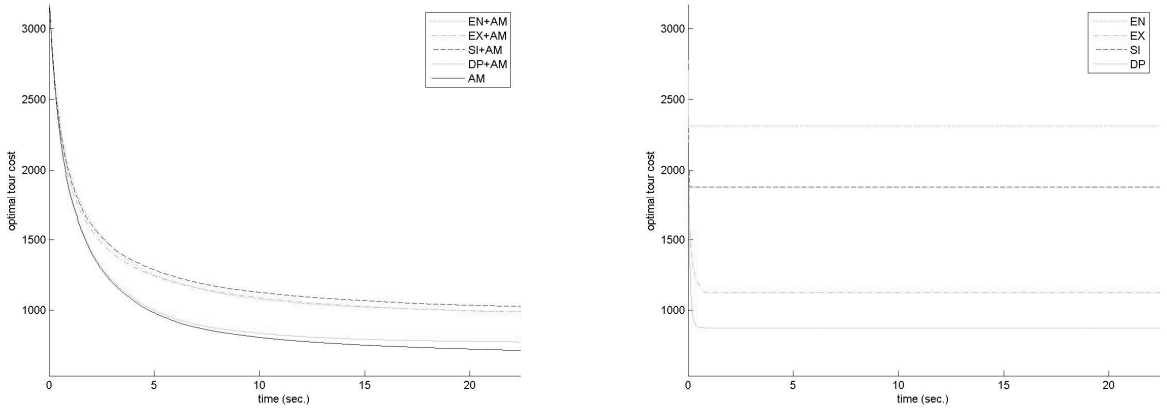
(a) (b)

Figure 7. Averaged convergence curves of Bayg29 under different mutation operators: (a) NrGA and (b) standard GA.



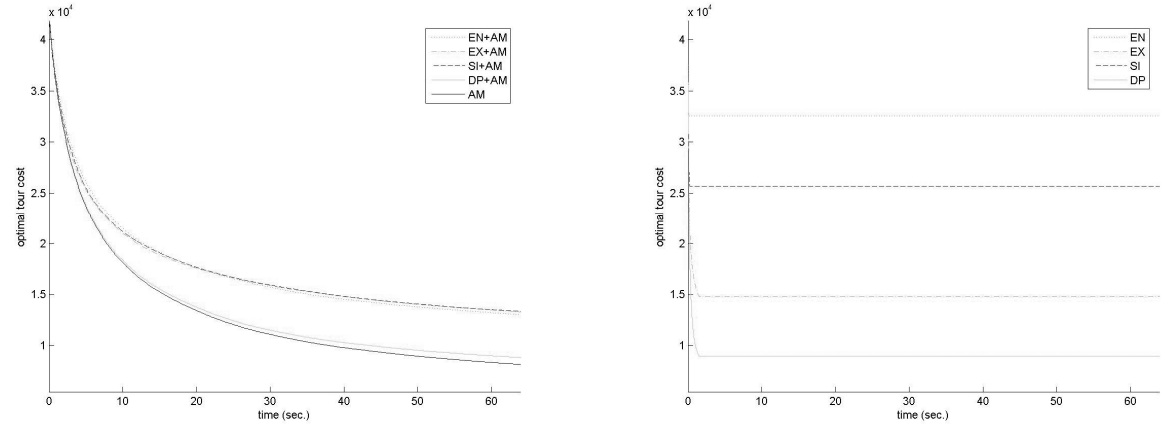
(a) (b)

Figure 8. Averaged convergence curves of Gr48 under different mutation operators: (a) NrGA and (b) standard GA.



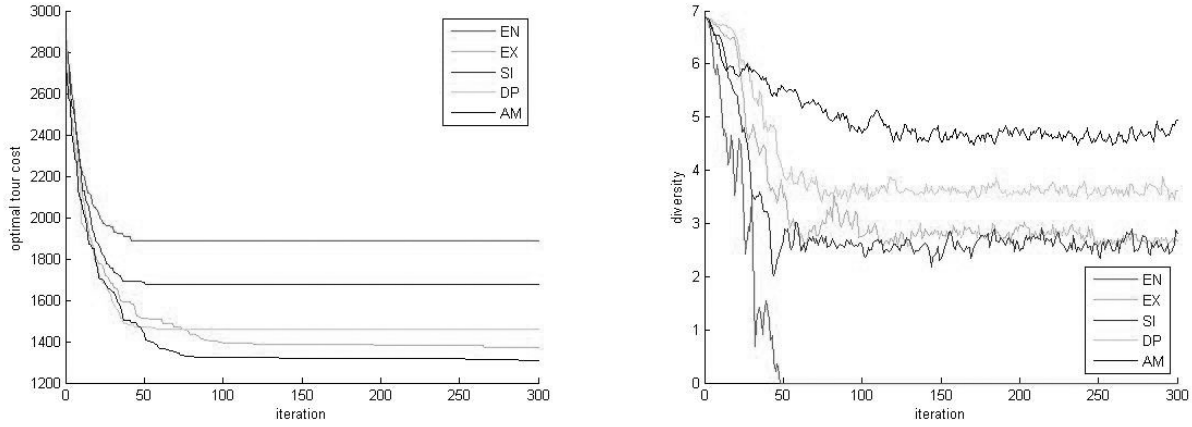
(a) (b)

Figure 9. Averaged convergence curves of ST70 under different mutation operators: (a) NrGA and (b) standard GA.



(a) (b)

Figure 10. Averaged convergence curves of Ch130 under different mutation operators: (a) NrGA and (b) standard GA.



(a) (b)

Figure 11(a) The optimal tour costs of Gr24 against iterations and (b) the diversities of NrGA and GA of Gr24 against iterations.