Self-Crossover and Its Application to the Traveling Salesman Problem

Malay K. Kundu and Nikhil R. Pal

Machine Intelligence Unit
Indian Statistical Institute
203 B. T. Road,
Calcutta 700035, INDIA
e-mail: { malay,nikhil}@isical.ac.in

Abstract. Crossover is an important genetic operation that helps in random recombination of structured information to locate new points in the search space, in order to achieve a good solution to an optimization problem. The conventional crossover operation when applied on a pair of binary strings will usually not retain the total number of 1's in the offsprings to be the same as that of their parents. But there are many optimization problems which require such a constraint. In this article, we propose a new crossover technique called, "self-crossover", which satisfies this constraint as well as retains the stochastic and evolutionary characteristics of genetic algorithms. We have also shown that this new operator serves the combined role of crossover and mutation. We have proved that self-crossover can generate any permutation of a given string. As an illustration, the effectiveness of this new operator has been demonstrated in solving the traveling salesman problem (TSP) using GA. This new technique is best suited for path representation of tours and performs better for TSP with large number of cities. Performance of the proposed scheme is compared with that of ordered crossover (OC) scheme.
Keywords : Genetic Algorithms, Heuristic Searching, Self-crossover, Traveling Salesman Problem(TSP), Ordered Crossover.

1 Introduction

The TSP, a well-known NP-hard problem, can be easily stated as follows : A traveling salesman must visit every city in his territory exactly once and then return to the starting point. An itinerary has to be found such that the total distance traversed in the tour is minimum. Mathematically, given a sequence of cities $c_1, c_2, \ldots, c_n$ and intercity distances $d(c_i, c_j)$, TSP finds a permutation π of the cities that minimizes the sum of distances

$$\text{TC(tour)} \;=\; \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \;+\; d(c_{\pi(n)}, c_{\pi(1)}). \tag{1}$$

We considered, $d(c_i, c_j) \;=\; d(c_j, c_i)$ for $1 \leq i, j \leq n$.

I. Imam et al. (Eds.): IEA/AIE-99, LNAI 1611, pp. 326–332, 1999.

During the last decades, several algorithms emerged like nearest neighbor, greedy algorithm, minimum spanning tree [4] to approximate the optimal solutions for problems. Another group of algorithms (like 2-opt algorithm, Lin-Kernighan algorithm) aims at a local optimization - an improvement of a tour by local perturbations. But none of the heuristics is well suited for large TSPs (*i.e.* TSP with large number of cities).

Genetic algorithms (GAs) are probabilistic heuristic search processes based on natural genetic system. They are capable of solving a wide range of complex optimization problems using three simple genetic operations (selection/ reproduction, crossover and mutation) on coded solutions (strings/ chromosomes) for the parameter set, not the parameters themselves, in an iterative fashion. There are several interesting features which made GA very popular. GAs consider several points in the search space simultaneously, which reduces the chance of convergence to a local optima. GAs use only the payoff or penalty function (objective function) called, the fitness function and do not need any other auxiliary information.

Eventually, TSP also became a target for GA applications. TSP tours can have various representations. Some of them are 1) adjacency 2) ordinal 3) path and 4) binary matrix representations. Based on different representations and choices of genetic operators several GA-based algorithms have already been reported [2, 3].

In this note we propose a new genetic operator, called *self-crossover* (SC). This operator is capable of randomly permuting the entries of a GA chromosome. If the chromosome is a binary string, in absence of mutation, this operator is able to retain the number of 1's in the string same, before and after the self-crossover operation. So, this operator can be successfully applied to a group of problems like selection of a fixed number of features, selection of a fixed number of prototypes for designing a nearest neighbor (NN) classifier where the constraint on total number of 1's in the chromosome string is very important. We have shown that self-crossover can produce any arbitrary string from any arbitrary starting chromosome. Hence self-crossover alone (*i.e.*without mutation) is sufficient for GA to be applicable to TSP and some other problems[7].

2 Review of GAs for TSP

In our investigation, we use the path representation of tours. In path representation, a tour $5 - 1 - 7 - 8 - 9 - 4 - 6 - 2 - 3$ is represented simply as (5 1 7 8 9 4 6 2 3).

There are several crossover techniques applicable to path representation. Order crossover (OC) of Davis [5], builds offspring by choosing a subsequence of a tour from one parent and preserving the relative order of cities from the other parent.

Recently a non-vector representation scheme for TSP has been evolved [1]. In this scheme, tours are represented by a *binary matrix* M. Matrix element m_{ij} contains a 1 if and only if the tour goes from city i directly to city j. This means

that there is only one non-zero entry for each row and each column in the matrix. This representation avoids the problem of specifying the starting city. However, not every matrix with these constraints would represent a single valid tour.

Homaifar and Guan [1] proposed a new crossover technique based on matrix representation, called matrix crossover (MC). MC exchange all entries of the two parent matrices after a crossover point (which represents a column number of matrices). An additional "repair algorithm" is run to ensure that each row and each column has precisely single 1; and to cut and connect sub-tours to produce a single legal tour. The first step of the"repair algorithm" moves some 1's in the matrix to satisfy the row and column constraints. The cut and connect phase takes into account the existing edges in the original parents, preserving as many of the existing edges from the parents as possible. Homaifar and Guan [1] incorporated *inversion* operator to bring the effect of mutation in the GA. Inversion operation needs the path representation of the tours and selects an arbitrary subsequence of cities from the parent tour. The order of this subsequence of cities is reversed producing a new tour.

Though it is true that the order of cities (not the positions of the cities) are important for tours, matrix crossover using matrix representation is not an elegant way to evolve new tours. For mending the illegal tours some repair algorithms are needed. Effectively, efficiency of those repair algorithms will determine the efficiency of GA-based heuristic algorithm for solving TSP.

MC alone is not able to evolve each and every corner of the total tour space. So some additional operators are needed to blend the flavor of mutation with it. In [1] authors considered inversion to play this role. For that they need to switch over to path representation from matrix representation and vice versa.

3 Self-Crossover(SC): A New Genetic Operator

Unlike Conventional crossover mechanism, self-crossover mechanism alters the genetic information within a *single* potential string selected **randomly** from the mating pool to produce an offspring. This is done in such a manner that the stochastic and evolutionary characteristics of GAs are preserved.

Let $S = 00010010011001011011$ be a string of length 20 selected from the mating pool. For self-crossover, first we select a random position p $(0 < p < L)$ and generate two substrings s_1 and $s_2 : s_1 =$ bits 1 through p of S and $s_2 =$ bits $p+1$ through L of S. Now we select two random positions p_1, $0 \leq p_1 \leq p$ and p_2, $0 \leq p_2 \leq (L-p)$. Then four substrings are generated as follows :

$$s_{11} = \text{bits 1 through } p - p_1 \text{ of } s_1$$
$$s_{12} = \text{bits } (p - p_1 + 1) \text{ through } p \text{ of } s_1$$
$$s_{21} = \text{bits 1 through } L - p - p_2 \text{ of } s_2$$
$$s_{22} = \text{bits } (L - p_2 + 1) \text{ through } L \text{ of } s_2$$

Using operations similar to crossover we generate $S^1 = s_{11} \mid s_{22}$ and $S^2 = s_{21} \mid s_{12}$. Finally, the self-crossovered offspring of S is generated as $S_1 = S^1 \mid S^2$. It is easy to see that number of 1's in S and S_1 is the same. We now explain it

with the example string S of length 20.

$$S = 00010010011001011011$$

A random position, $p = 9$, is selected for splitting the string into two substrings (s_1, s_2) as follows : $s_1 = 000100100$ and $s_2 = 11001011011$

Now two random positions, $p_1 = 4$ and $p_2 = 7$, are selected for s_1 and s_2 respectively. After splitting s_1 and s_2 at $4th$ and $7th$ position, respectively we get,

$s_{11} = 00010$, $s_{12} = 0100$, $s_{21} = 1100$, and $s_{22} = 1011011$.

The two new substrings S^1 and S^2 are then obtained as :

$S^1 = 000101011011$ and $S^2 = 11000100$.

Finally, the offspring (S_1) is generated by concatenating S^1 and S^2 as :

$S_1 = 00010101101111000100$

Thus, self-crossover exchanges substrings s_{12} and s_{22}. If the parent string consists of all 0's or all 1's, the offspring generated through self-crossover will resemble its parent because of the underlying constraint on the total number of 1's in the string. It is also clear that if we do not start GA with a all '1' or all '0' string, GA with self-crossover technique, will never generate such strings as offsprings. So, self-crossover will generate new offsprings as iterations go on.

We can see very well that mutation is not effective in producing such constrained offsprings. But self-crossover can regenerate any lost genetic information. So we may not need mutation when we use the new technique in constrained GA applications.

Next we show through a Lemma that self-crossover (without mutation) can generate any target string.

Lemma : Given a string of symbols, self-crossover operations can generate any arbitrary permutation of the symbols.[7]

Proof : We can represent any arbitrary string P by $P = S_1 \mid S_2 \mid S_3 \mid S_4$ where S_i represents a subsequence of symbols. S_i can be empty sequence as well. Now if we are able to prove that a parent string $P = S_1 \mid S_2 \mid S_3 \mid S_4$ can produce an offspring $O = S_1 \mid S_3 \mid S_2 \mid S_4$ using a finite number of self-crossover operations, then we can iterate the process to cook up a sequence of self-crossover operations to reach any target offspring.

The position for splitting P is chosen such that two subsequence s_1 and s_2 are formed as $s_1 = S_1 \mid S_2$ and $s_2 = S_3 \mid S_4$. Now two random positions for s_1 and s_2 are selected such that after splitting of s_1 and s_2 at these two positions we get

$s_{11} = S_1$, $s_{12} = S_2$, $s_{21} = S_3$, and $s_{22} = S_4$. So the resultant child is obtained as

$P_1 = S_1 \mid S_4 \mid S_3 \mid S_2$.

We apply once more the self-crossover operation on intermediate child P_1. Now the random position for splitting P_1 is chosen such that

$s_1 = S_1 \mid S_4$ and, $s_2 = S_3 \mid S_2$.

Again we select two random positions for s_1 and s_2 such that after splitting of s_1 and s_2 at these two positions we get

$s_{11} = S_1$, $s_{12} = S_4$, $s_{21} = S_3 \mid S_2$, and s_{22} = empty sequence.
The offspring now becomes
$O = S_1 \mid S_3 \mid S_2 \mid S_4$ which is nothing but what we wanted to produce.

Since S_1 could be a null string, so any symbol from the parent string can be brought at the beginning of the offspring through substring S_3 by two successive self-crossover operations. The lemma also ensures that any substring consisting of symbols starting from the beginning of a parent string can be preserved in the child through substring S_1. Hence, any target permutation can be grown from the left side. Proceeding this way in the terminal phase of the process S_2 and S_4 will be empty; S_1 will contain the entire target substring except the last symbol which will be in S_3.

Note that, the lemma does *not* say that there is no more need for mutation in GA with self-crossover technique. It simply says that for combinatorial problems like TSP, use of self-crossover without mutation can generate all possible valid solution strings. For problems like feature selection[7], data editing for NN classifier where we want to select the best subset of features or data points of a prefixed cardinality, self-crossover without mutation is sufficient. In fact, conventional mutation for such problems may produce invalid solutions, *i.e.*, it may generate a substring of arbitrary cardinality, not equal to the prefixed cardinality.

At the first sight, it might appear that self-crossover is nothing but a parallel random search, but this is not the case because of two reasons. Self-crossover is done only on a randomly selected subset of strings and self-crossover does not alter the substring s_{11}. It exchanges, only s_{22} and s_{12}. Consequently, the evolutionary characteristics of GA are preserved. The similarity between the parents and offsprings will be more if we take $p_1 = p_2 = p'$ (say) = a random number selected between 1 and $Min\ (p, L-p)$; i.e., $0 < p_1 = p_2 = p' < Min\ (p, L-p)$. Here, the bits in positions 0 through p' and in positions $p+1$ through $L-p'$ will remain unaltered. Consequently, the evolution pressure will be high.

4 TSP with Self-Crossover(SC)

It is seen that the SC operator plays the combined role of conventional crossover and mutation. So, we are getting two-in-one effect with the help of this operator. Because of the simplicity of this operator, our algorithm becomes fast. Here we use path representation of tours. Since the SC operator can generate any permutation of a given string, as already discussed, given a valid tour, it will generate only valid tours. Now we describe the algorithm for solving TSP. The objective here is to minimize the total cost of a tour, TC(tour). To convert it to a maximization problem, we take the fitness function

$$f(tour) = - \text{TC(tour)}.$$

1. Start with the population of initial valid tours(a set of integer strings/ chromosomes).
2. Evaluate fitness of every tour in the current population.
3. Generate new mating pool .

4. Each tour of the current population is self-crossovered and copied to the new population.
5. Repetition of steps 2 through 4 until the system ceases to improve or some stopping criterion is reached.

5 Results and Discussion

The results obtained for TSPs of different sizes are depicted in table I. For each problem we synthetically generated a cost matrix with known cost for the optimum tour. The initial population is generated randomly, consisting of only valid tours. We adopted partially disruptive selection strategy *i.e.*, at each iteration we kept few best candidates and chose the rest of the population randomly.

Table : I

Result with SC				
Cities	Population size	Generations	Best tour-cost	Optimum cost
10	10	2876	49.45	49.45
20	20	99943	36.74	36.74
30	30	151987	35.349	34.349
50	50	201000	7.609	5.98

From table I we found that GA equipped with SC can determine exact optimum solution for TSPs of sizes 10 and 20 within few generations. For TSP with 30 cities, our scheme can find a solution within 3% of the optimum solution.

Note that, in table I we used different population sizes for different problems of different sizes. For TSP the population size should be chosen carefully, it should be dependent on the size of TSP; larger the size of TSP, larger should be the population size.

SC(Self-crossover) operation does not involve any comparison operation which is an explosive cpu operation. It needs to choose 3 random numbers and a few string copy and string concatenation operations. But OC(Ordered-crossover) operation needs 2 string comparisons per iteration. String comparison operation cost is once again proportional to the string length. Unlike SC, number of OC operations per iteration is proportional to the square of the population size since OC involves a pair of strings (Number of all possible pairs $=n*(n-1)/2$ where n represents the population size). As a result the time complexity for GA with OC is much more than GA with SC. The time complexity for GA with OC increases even more prominently for higher number of cities, compared to GA with SC. As an illustration, OC needs 17 minutes for 5000 iterations for 20 cities, whereas SC requires only 50 seconds for 5000 iterations for same problem.

6 Conclusions

There is a class of optimization problem which require the number of '1's in the strings to be constant. Conventional crossover / mutation does not guarantee

this. We have introduced a new crossover operation named *self-crossover* which preserve this constraint. GA with SC has been successfully used for selection of a fixed number of good features for pattern recognition[7]. Here GA with SC is used for study the TSP. SC has some distinct advantages over other GA based approaches for TSP. For example SC produces only legal tours and prevents generation of duplicate tours. Moreover, this new operator plays the combined role of mutation and as well as conventional crossover operator. GA with SC is found to be quite successful for TSP of moderate size. However experiments with problems of much bigger sizes are needed to make finer conclusion.

7 References

1. Homaifar, A. Guan, S. and Liepins, G. E. : A New Approach on the Traveling Salesman Problem by Genetic Algorithms, 460-466. To appear in *Complex Systems*.
2. Grefenstette, J. J., *et al.*, Genetic Algorithm for the TSP, in Grefenstette (ed), *Proceedings of an International Conference of Genetic Algorithm and their Applications,* Texas Instrument and U.S. Navy Center for Applied Research and Artificial Intelligence, 154-159, (1985).
3. Suh, *et al.*, The Effects of population size, Heuristic Crossover and Local Improvement on a GA for the TSP, in the J.D.Schaffer (ed), *Proceedings of the Third International conference for Genetic Algorithms*, Morgan Kaufmann Publishers, Inc., 110-115, (1989).
4. Johnson, D. S., Local Optimization and Traveling Salesman Problem, in M.S. Paterson (Editor), *Proceedings of the 17th Colloquium on Automata, Languages, and Programming, Springer-Verlag, Lecture Notes in Computer Science*, **443**, 446-461, (1990).
5. Davis, L., Applying Adaptive Algorithms to Epistatic Domains, *Proceedings of the International Joint Conference on Artificial Intelligence*, 162-164, (1985).
6. Lawler, E. *et al.*, The Traveling Salesman Problem, *John Wiley and Sons*, New York.
7. , Pal, N. R. and Nandi, S. and Kundu, M. K. : Self Crossover: a new genetic operator and its application to feature selection, *International Journal of System Sciences*, **29**, no. 2, 207-212, (1998).