New Operators of Genetic Algorithms for Traveling Salesman Problem

Shubhra Sankar Ray, Sanghamitra Bandyopadhyay and Sankar K. Pal
Machine Intelligence Unit
Indian Statistical Institute
Kolkata 700108
{shubhra_r, sanghami, sankar}@isical.ac.in

Abstract

This paper describes an application of genetic algorithm to the traveling salesman problem. New knowledge based multiple inversion operator and a neighborhood swapping operator are proposed. Experimental results on different benchmark data sets have been found to provide superior results as compared to some other existing methods.

Keywords: knowledge based multiple inversion, order crossover, knowledge based neighborhood swapping.

1. Introduction

The Traveling Salesman Problem (TSP) is one of the top ten problems, which has been addressed extensively by mathematicians and computer scientists. Its importance stems from the fact there is a plethora of fields in which it finds applications e.g., DNA fragment assembly, VLSI design. The classical formulation is stated as: Given a finite set of cities and the cost of traveling from city *i* to city *j*, if a traveling salesman were to visit each city exactly once and then return to the home city, which tour would incur the minimum cost? Formally, the TSP may be defined as follows [1]:

Let *{1, 2, ... n}* be the labels of the *n* cities and $C = [c_{i,j}]$ be a $n \times n$ cost matrix where $c_{i,j}$ denotes the cost of traveling from city *i* to city *j*. The total cost *A* of a TSP tour is given by

$$A(n) = \sum_{i=1}^{n-1} C_{i,,i+1} + C_{n,1} \qquad (1)$$

The objective is to find a permutation of the *n* cities which has minimum cost. The TSP is a very well known NP-hard problem [2] and therefore any problem belonging to the NP-class can be formulated as a TSP problem.

Over decades, researchers have suggested a multitude of heuristic algorithms, including genetic algorithms (GAs) [3], for solving TSP [6]. In this article we propose some new operators, namely Knowledge Based Multiple Inversion (KBMI) and Knowledge Based Neighborhood Swapping (KBNS) along with a Modified Order Crossover for solving TSP. The experimental results obtained on TSP benchmarks have been found to be superior in terms of quality of solution when compared with other existing GAs [6].

2. Proposed GA for TSP

A new algorithm, called SWAP_GATSP, is described in this section for solving TSP using elitist GAs with new operators namely, Knowledge Based Multiple Inversion, Modified Order Crossover and Knowledge Based Swapping. The structure of the proposed SWAP_GATSP is presented below.

```
begin SWAP_GATSP
      Create initial population of tours randomly.
       while generation_count < k do
              /* k = max. no. of generations.*/
            begin
                KBMI
                Natural selection
                MOC
                KBNS
                Mutation
                Elitism
                Increment generation_count.
            end ;
            Output the best individual found.
end SWAP_TSP.
```

2.1. String representation and Cost function

In order to find the shortest tour for a given set of *n* cities using GAs, the path representation [6] is more natural for TSP and has been well studied. In this encoding, the string representation for a TSP tour is an array of n integers which is a permutation of {1, 2, …… n}. The objective is to find a string with minimum cost. In the following subsections the new genetic operators employed in the proposed GA are described.

2.2. Knowledge Based Multiple Inversion

In this process for each string the distance between every two consecutive cities is calculated from the cost matrix and the distances are sorted in descending order. A record is kept so that one can find which distance corresponds to which two cities.

Suppose for a string (1 2 3 4 5 6 7 8 9) the sorted distances are between cities

(1,2), (5,6), (3,4), (4,5), (9,1), (2,3), (8,9), (6,7) and (7,8).

Now the substring between the highest two distances (1,2) and (5,6) is inversed, resulting in the parent (P) and the child (C) as follows

P1 = (1 |2 3 4 5 |6 7 8 9)
and
C1 = (1 |5 4 3 2 |6 7 8 9)

This inversion procedure is repeated for pairs [(3,4) and (4,5)], [(9,1) and (2,3)], [(8,9) and (6,7)] and so on for string with higher no of cities with the condition that a inversion process will not take place if a substring for a pair overlaps any other substring of previous all the pair. Now the substring for the 2nd pair [(3,4) and (4,5)] overlaps the substring for the pair [(1,2) and (5,6)], so no inversion of string will take place for the pair [(3,4) and (4,5)] and the pair will be removed from the list. The resulting list then becomes

[(1,2) and (5,6)], [(9,1) and (2,3)] and [(8,9) and (6,7)]

The substring for the pair [(9,1) and (2,3)] now overlaps with the substring for the pair [(1,2) and (5,6)], again no inversion of string will take place. The resulting list is

[(1,2) and (5,6)] and [(8,9) and (6,7)]

Now for the pairs [(1,2) and (5,6)] and [(8,9) and (6,7)] there is no overlap between substrings. Therefore the modified child C2 obtained from C1 are as follows

C1 = (1 |5 4 3 2 |6 |7 8| 9)
and
C2 = (1 5 4 3 2 6 |8 7| 9)

Regarding the number of pairs (say, pa) to be taken for the 1st iteration and for number of cities within 100, we have found experimentally that pa's are 3, 3, 4, 5 and 7 for number of cities 24, 29, 48, 70 and 100 respectively. These values can be achieved by an equation of the form

pa=[(n+32)/20]

where n is the number of cities.

This is not kept constant over the generations, rather it is varied in cycles of appropriate intervals linearly from

1) [pa] to 0.0 for iteration 1 to [z/3]
2) 0.0 to [pa] for iteration [z/3] to [2×z/3]
3) [pa] to 0.0 for iteration [2×z/3] to [z]

where z is the total no. of iterations performed in one run.

This kind of variation helps in exploring the search space efficiently and prevents the GA from getting stuck in the local optima. Note that this is an upgraded version of Simple Inversion Mutation [6], which is discussed later. But, it can't be called mutation operator, as it is a decisive process not a random one.

2.3. Natural Selection

This operator is designed by a common method of natural selection in GA called the Roulette Wheel method [3]. The Roulette Wheel method simply chooses the strings in a statistical fashion based solely upon their relative (i.e., percentage) cost or fitness values. So, the natural selection operator in this GA randomly chooses strings from the current population with probability inversely proportional to their cost.

2.4. Crossover

Before presenting our new crossover strategy, a closely related existing method known as Order based crossover is described briefly. It has been observed to be one of the bests in terms of quality and speed, and yet is simple to implement.

Order Based Crossover (OBC). The order based crossover operator [7] selects at random several positions in one of the parent tours, and the order of the cities in the selected positions of this parent is imposed on the other parent to produce one child. The other child is generated in an analogous manner for the other parent.

Modified Order Crossover. A randomly chosen crossover point divides the parent strings in left and right substrings. The right substrings of the parents s1 and s2 are selected. After selection of cities the process is the same as the order crossover. Only difference is that instead of selecting random several positions in a parent tour all the positions to the right of the randomly chosen crossover point are selected.

For example with the following parents and crossover point

s1 = (1 2 3 4 |6 9 8 5 7)
and
s2 = (2 1 9 8 |5 6 3 7 4),

after position selection

s1 = (1 2 * * * 9 8 * *)
and
s2 = (2 1 * * * * 3 * 4)

we obtain the generated pair of children as

b1 = (1 2 5 6 3 9 8 7 4)
and
b2 = (2 1 6 9 8 5 3 7 4)

Clearly this method allows only the generation of valid strings.

2.5. Knowledge Based Neighborhood Swapping

This is a novel deterministic operation injected in the typical structure of elitist GA. This works on the set of all strings obtained from crossover. For a string s in the population a random index value i is generated (1<i<S , where S is string length). Now for city (i-1) and city (i+1) find the city that is nearest to both the cities. Let its index be j. In this search the cities (i-1) and (i+1) are excluded. Then in s, swap the city at ith position with that at j^{th} position. This operation is repeated for all the strings in the population.

Index j is obtained as follows:
1) calculate the distance from the cost matrix for a particular city (say c) as
X(c)= distance((i-1), c) + distance((i+1), c)
2) repeat step 1 for all the cities except city(i-1) and city(i+1)
3) find the city c for which X(c) is minimum and make j=c
Since it is a deterministic operator based on the cost matrix, it helps the stochastic environment of the working of GA to derive an extra boost in the positive direction. As this operator is applied only once on every string for a random city index, not on all the cities in that string, this operation is not expensive.

2.6. Mutation

For the TSP the simple inversion mutation (SIM) and insertion mutation (ISM) are the leading performers [6]. Here simple inversion mutation (SIM) is performed on each string as follows:
This operator selects randomly two cut points in the string, and it reverses the substring between these two cut points. For example consider the tour
(1 2 3 4 5 6 7 8)
and suppose that the first cut point is chosen randomly between 2^{nd} city and 3^{rd} city, and the second cut point between the 5^{th} city and the 6^{th} city. Then the resulting strings will be
P = (1 2 |3 4 5| 6 7 8)
C = (1 2 |5 4 3| 6 7 8)

The mutation probability it is not kept constant over the generations. Rather it is varied in cycles of appropriate intervals [1] (linearly from 0.06 to 0.003 where n is the number of cities).

3. Time complexity of proposed GA

The time complexity of the algorithm SWAP_GATSP is given by O(k•N•n) where k is the number of generations, N is the population size and n is the data size or the number of cities.

4. Experimental Results

SWAP_GATSP was implemented in Matlab 5.1 on Pentium-4 (1.7 GHz) and the results were compared with those obtained from the survey of Larranaga [6] and [1]. Results are also compared with a public domain TSP solver based on GA by Michael Lalena [5] having the proclamation of being the fastest among known solvers.

Table 1 summarizes the final results obtained by running the Multiple Inversion GA on several symmetric TSP instances containing 24, 29, 48, 70 and 100 cities, taken from the TSPLIB [4]. The best results from 30 run are listed here. The number of populations is taken 10 for Grtschels24.tsp and bayg29.tsp. The population is 24 for 48 cities, 30 for 70 cities and 40 for 100 cities. Crossover probability was fixed at 0.85 across the generations. As discussed in Section 2.6, the mutation probability was varied linearly in with iteration, maximum being 0.06 and minimum 0.003. These values are experimentally obtained which gives very good results.

For Grtschels24.tsp the previous best results for other GAs were 1272 km [6]. For Grtschels48.tsp the previous best result of 5074 km was with ER crossover and SIM mutation [6]. These investigations were carried out with population size of 200, mutation probability 0.01 and 50000 iterations [6]. The proposed approach exceeds the previous best result as shown in Table 1 with less no. of population size and iteration. For st70.tsp our result is compared with GA of Lalena with best result of 895 km. As Lalenas software is not downloadable at this instant for some difficulties in his web site, we compared the result with that in [1] where Lalenas GA was downloaded at that time. In [1] it was stated that the best result for their algorithm is 776 km for st70.tsp with population size of 50 and 5000 iterations. GA with only order crossover and simple inversion mutation (OX-SIM) is implemented next as standard GA for TSP [6, 1] for all the problems and the results are compared with SWAP_GATSP.

Table 1

		Best results for different TSPs				
Problem	Optimal	Proposed	[5]	[1]	Best GA in [6]	OX-SIM
Grtschels24	1272	1272 (500 iter)	----	----	1272	1272 (8,000 iter.)
bayg29	1610	1610 (600 iter)	----	----	----	1620 (10,000 iter.)
Grtschels48	5046	5046 (800 iter.)	----	----	5074	5097 (12,000 iter.)
St70	675	685 (2000 iter.)	895	776	----	888 (15,000 iter.)
KroA100	21282	21504 (5000 iter.)	----	----	----	22,400 (25,000 iter.)

Table 2

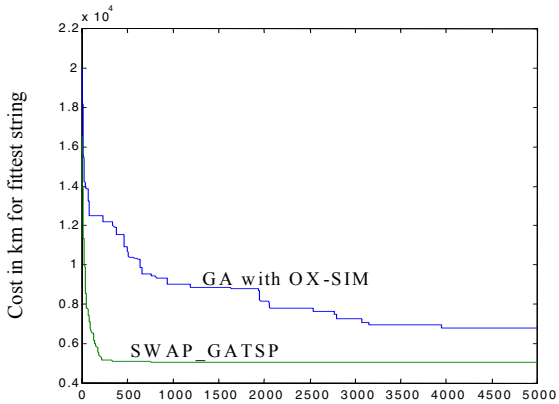
Problem	Proposed GA	Best GA in [6]	OX-SIM
Grtschels24	1272	1274	1342
bayg29	1615	-----	1720
Grtschels48	5110	5154	5451
St70	710	-----	920
KroA100	21,900	-----	23,200

Table 2 shows the average results after 5000 iterations. Here OX-SIM is implemented, but the average results for best previous GA are taken from Larranaga [6].

The SWAP_GATSP and the GA with OX-SIM have been compared w.r.t. computation time. Both programs were run for 358 seconds for Grtschels48.tsp and the fitness of fittest string is plotted with iteration as shown in Fig 1. In 358 seconds the SWAP_GATSP has gone through 3600 iterations and the GA with OX-SIM run for 5000 iterations. The lower graph shows that the optimal cost of 5046 km is achieved within 800 iterations for the SWAP_GATSP whereas cost is 6773 km for GA with OX-SIM (shown in upper graph). Similar results are also found when the proposed method is compared with other GAs stated in [6].

5. Conclusion

The results obtained with the newly designed genetic operators in our algorithm are impressive, on practical data set. Larger benchmarks are to be tested next. This method can be easily adapted to solving the asymmetric TSP. Experiments on comparing those results with other existing solvers for asymmetric TSP also need to be performed. Application of the developed SWAP_GATSP to real life problems like DNA fragment assembly, an important issue in bioinformatics, should be studied. The authors are currently working in this direction.



No of iterations

Figure 1:Cost of fittest string Vs. Iteration for Grtschels48.tsp

References

[1] Sur-Kolay S., Banerjee S., and Murthy C. A., "*Flavours of Traveling Salesman Problem in VLSI Design*", 1st Indian International Conference on Artificial Intelligence, *2003*.

[2] Garey, M. R., and Johnson, D. S.: *"Computers and Intractability: A Guide to the Theory of NP-completeness*", W. H. Freeman and Co., San Francisco, 1979.

[3] Goldberg, D. E.: "*Genetic Algorithm in Search, Optimization and Machine Learning",* Machine Learning. Addison-Wesley, New York, 1989.

[4] TSPLIB Homepage: http://www.iwr.uniheidelberg.de/groups/comopt/software/TSPLIB95/

[5] Lalena, M.: TSP solver. http://www.lalena.com/ai/tsp

[6] Larranaga, P., Kuijpers, C. M. H.,Murga, R. H., Inza, I., Dizdarevic, S.: "*Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators",* Artificial Intelligence Review. 13, 1999, 129-170.

[7] Syswerda, G, "*Schedule optimization using genetic algorithms, Handbook of Genetic Algorithms*", Van Nostrand Reinhold, New York, 1991, 332-349.