A New Genetic Approach for the Traveling Salesman Problem

Thang N. Bui[†] and Byung R. Moon[‡]

***Abstract*— A new genetic algorithm (GA) for the Traveling Salesman Problem (TSP) is given. Two novel features of this algorithm are: (i) a new locus-based encoding/crossover pair, and (ii) a static preprocessing step which changes the encoding order of the vertices. It is believed that this algorithm is also applicable to other ordering problems, not just TSP. Experimental results on the standard benchmarks for TSP are favorable.**

I. Introduction

Let $G = (V, E)$ be a complete graph with weights on the edges. A *Hamiltonian cycle* of G is a cycle that visits each vertex of the graph exactly once. The *Traveling Salesman Problem* (TSP) is the problem of finding a Hamiltonian cycle with minimum weight. TSP is well-known to be NP-hard [6]. It has been extensively studied in the past for its wide applications as well as for its complexity which is representative of a difficult optimization problem [15]. Genetic algorithms have been applied to TSP with varying degree of success [8], [10], [16], [13], [22], [1], [21], [18], [12].

The output of an algorithm for TSP is a cyclic ordering of the vertices. Thus it is not unusual to see that most of the known genetic algorithms for TSP use order-based encoding schemes where the enumeration of the vertices in a cycle constitutes a chromosome. It is, however, also possible to use a locus-based encoding for TSP. The choice of encoding scheme directly affects the crossover operator. In fact, crossover operators can also be grouped into order-based ones and locus-based ones.

Locus-based encodings have some advantages over order-based ones: (i) there are no duplications for the same cycle as in the case of order-based encodings, which gives GAs a simpler view of chromosomes, and (ii) traditional crossover[1] is applicable. But locus-based encodings have an intrinsic weakness: roughly speaking, schemata consisting of widely scattered specific positions have poor survival probability through crossovers due to their long defining lengths or their almost uniform distribution of specific symbols [3], [4]. This phenomenon is not restricted to just TSP. On the other hand, order-based encodings do not have any fixed position assignment, consequently they can utilize vertices' geographical linkage (see [7], pp. 169–178).

Most order-based crossover operators cannot effectively use adjacency information contained in the two parents [22]. Whitley et al devised a crossover operator, Edge Recombination (ER), which is capable of preserving adjacency information by concentrating on edges in the two parents [22], [18]. ER turned out to have a better performance than most order-based crossovers on TSP [18]. This paper aims to devise a new traditional type *locus-based* crossover operator for TSP which is capable of effectively using adjacency information as ER and at the same time remedies the inherent weakness of other locus-based encodings in schema preserving capability mentioned in the previous paragraph.

It has been observed that the performance of GAs on problems with a locus-based encoding can be improved by reordering the indices of the vertices, i.e., by preprocessing. The technique of preprocessing for genetic algorithms was first suggested in [2], [3] on unweighted sparse graphs. The basic idea of preprocessing is to carefully reassign the loci of vertices in chromosome encodings to help GAs effectively preserve good schemata. In this paper, we extend the technique to weighted graphs.

We assume the reader is familiar with the combinatorial notations $\Theta(\cdot)$, $O(\cdot)$ and $\Omega(\cdot)$ for time complexity, and the standard terminologies of GA: chromosome, gene, schema, specific symbol, defining position and defining length. As cities of TSP instances correspond to vertices in graphs, we use the terms vertex and city interchangeably.

[†]Department of Computer Science, Pennsylvania State University at Harrisburg, Middletown, PA 17057, USA.
[‡]Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16801, USA.

[1]Choosing a couple of crossover points and alternately copying values from two parents interval by interval.

II. A New Crossover Operator

A. Motivation

It was Grefenstette who first suggested a crossover operator using the four edges that are incident to a vertex in the two parent Hamiltonian cycles [9]. Whitley et al devised a crossover operator called Edge Recombination (ER), which is based on Grefenstette's but has a better edge-preserving property [22], [18]. To effectively use incident edges of each vertex, it generates an array of n elements for n vertices and each of them has up to four incident edges for each vertex in the two parents. In other words, link information is extracted separately before every crossover is performed. Their work as well as Grefenstette's work is significant in the sense that it treats *edges* as the main objects of crossover. ER showed better performance than other order-based crossovers as mentioned in the introduction [18].

Recently Homaifar et al suggested an interesting crossover operator, Matrix Crossover (MX) [12]. MX uses an ordered representation. Before every crossover, MX generates two binary matrices from the two parents representing the cycles. Crossover is performed on the two matrices and a new matrix is produced. This new matrix is transformed into an ordered representation, i.e., an offspring. For example, a parent chromosome *acbde* is transformed into the following binary matrix:

$$\begin{array}{c|c:cc:cc}
 & a & b & c & d & e \\
\hline
a & 0 & 0 & 1 & 0 & 0 \\
b & 0 & 0 & 0 & 1 & 0 \\
c & 0 & 1 & 0 & 0 & 0 \\
d & 0 & 0 & 0 & 0 & 1 \\
e & 1 & 0 & 0 & 0 & 0 . \\
 & & * & & * &
\end{array}$$

MX chooses one or two crossover point(s) at random separating the columns as shown above and copies the partitioned columns of the two parents alternately into the offspring. As the resulting matrix is usually not a valid Hamiltonian cycle an adjusting step is added. To save storage, it temporarily generates three matrices for a crossover operation instead of maintaining chromosomes in matrix form. But in any case the crossover dealing with matrices takes $\Omega(n^2)$ time where n is the number of vertices. This will be an unnecessary overhead if we can devise a linear sized encoding which is basically as powerful as the matrix representation. Homaifar et al's work is significant in the sense that it is a crossover operator which differs minimally from the traditional framework of locus-based crossover.

B. The New Crossover Operator

The main objective of this new design is to have a locus-based crossover pair like MX, but that takes only linear time and is able to effectively extract adjacency information as well as ER. The encoding is as follows. Each chromosome is a pair of strings, (S, P). The first part, the S string, of a chromosome consists of the successors of the cities in a Hamiltonian cycle, which has n genes. With this S string, it is basically possible to do all the works done by MX. With just the string S it takes constant time to find the successor of a city but $\Theta(n)$ time to find the predecessor, which would result in an $\Omega(n^2)$ time crossover operator. Thus, we use one more string, the P string, containing the predecessors of all the cities. So each chromosome has two strings each of size n for a total length of $2n$. Each city has a fixed locus on a chromosome and each locus has two values: one on the S string (its successor) and one on the P string (its predecessor). For example, the cycle *acbdea* is represented in this scheme by the (S, P) pair (*cdbea*, *ecabd*), where we have assumed that the loci of the cities in the chromosome are (*abcde*). For each cycle there is a unique representation under this scheme. Now it takes constant time for a city to find its successor or predecessor, which makes it possible to devise a linear time crossover operator. This dual chromosome includes all information in the extra data structure of ER. It is obvious that the P string can be directly generated by scanning the S string.

In the following, we suggest a linear time crossover operator which considers all possible adjacency information contained in parent cycles. It consists of three parts: copying S strings (Steps 1 and 2), adjustment (Steps 3, 4 and 5), and generating the P string (Step 6). Steps 1 through 5 are used to generate the S string. We select k random crossover points and let the $k + 1$ intervals created by the crossover points be labeled $0, 1, \ldots, k$. We refer to loci in even indexed intervals as even positions and the others as odd positions.

1. Copy the successor values of all odd positions of parent 1 into the offspring.
2. Copy the successor values of all even positions of parent 2 into the offspring. If a duplicated appearance with a value copied from parent 1 in Step 1 is detected, do not copy it.
3. Fill each unfilled locus, say city a, of Step 2 with one of the three remaining adjacent cities of a with the following priorities: the predecessor of a in parent 2, the successor of a in parent 1, the predecessor of a in parent 1.
4. Randomly fill in still unfilled loci with yet un-

used cities.

5. Resolve subcycles using the RESOLVE_CYCLES algorithm given in Figure 1. The result is the string S.
6. Make the P string by scanning the S string just produced.

Figure 1: Algorithm for resolving subcycles.

```
Algorithm RESOLVE_CYCLES(offspring, n)
  // cities are indexed by 0,1,2,...,n-1
  for i = 0 to n-1
    { visited[i] = UNVISITED; }
  starting_city = random() mod n;
  cycle1 = get_cycle(offspring, n, visited, starting_city);
  while (cycle1.no_cities < n) {
    while (visited[starting_city] == VISITED)   — (1)
       { starting_city = (starting_city + 1) mod n; }
    cycle2=get_cycle(offspring, n, visited, starting_city);
    merge_cycle(cycle1, cycle2, offspring, n);
    cycle1.no_cities = cycle1.no_cities + cycle2.no_cities;
  }
end RESOLVE_CYCLES.
```

Even when there is no duplicated gene value in an S string, it may not be a proper Hamiltonian cycle. It may consists of more than one mutually disconnected subcycles. Step 5 is for resolving this problem. In the RESOLVE_CYCLES algorithm, the starting city is selected at random. The routine get_cycle returns the cycle starting at *starting_city*. It also marks all cities in the cycle as VISITED. In merge_cycle, *cycle*2 is merged into *cycle*1. Eventually, all subcycles are merged into *cycle*1. In Step 3, if the local improvement heuristic is not so strong, it may be better to choose a city with the fewest number of unused edges as Whitley et al did in [18] for maximum edge preserving capability, which can be easily adapted in this model. Figure 2 shows an example of a 2-point crossover. Two parent cycles *adcefbga* and *aebcfdga* are used for instance. Assume two crossover points are chosen between loci b and c and between loci e and f.

Figure 2: Crossover scheme.

		*			*		
parent 1	*a*	*b*	*c*	*d*	*e*	*f*	*g*
successors	*d*	*g*	*e*	*c*	*f*	*b*	*a*
predecessors	*g*	*f*	*d*	*a*	*c*	*e*	*b*
parent 2	*a*	*b*	*c*	*d*	*e*	*f*	*g*
successors	*e*	*c*	*f*	*g*	*b*	*d*	*a*
predecessors	*g*	*e*	*b*	*f*	*a*	*c*	*d*
step 1:			*e*	*c*	*f*		
step 2:	[illegible]	[illegible]	*e*	*c*	*f*	*d*	*a*
steps 3 & 4:	*g*	*b*	*e*	*c*	*f*	*d*	*a*
step 5:	*c*	*g*	*e*	*b*	*f*	*d*	*a*
step 6:	*g*	*d*	*a*	*f*	*c*	*e*	*b*

C. Local Improvement

After an offspring is generated by crossover, it is locally improved with a scheme based on Lin-Kernighan local optimization (LK hereafter) [14]. Hybrid genetic algorithms have been considered natural in solving a difficult problem to get a desirable performance since genetic algorithms are not so good at fine tuning solutions near the optimum [9], [21]. The most common local improvement heuristics for TSP are 2-Opt [1], [13], [12], Or-Opt [1], [13], and Lin-Kernighan algorithm [21]. We tried all of these heuristics and could observe that combining 2-Opt and Or-Opt is visibly better than using just 2-Opt, and using LK is better than the latter when tuned within similar time. This is consistent with previous published results [13], [21]. 2-Opt and Or-Opt are $O(n^3)$ time algorithms but they typically take only $\Theta(n^2)$ time since the number of exchanges made is typically $O(n)$. The worst case running time of LK is also $\Theta(n^3)$. The original LK considers up to n-changes which is the most time consuming part. If we only consider up to k-changes, the time complexity becomes $\Theta(\max(k^2n, cn))$, where c is the number of cycle updates. In our algorithm we set k to be a fixed constant thus the time complexity depends on the number of cycle updates. Since c is at most $O(n)$, the worst case running time of our modified version of LK is $O(n^2)$. At early stages of the genetic algorithm, it is observed that c is usually $\Theta(n)$. But after generating some, say 100 or 200, offsprings, it rapidly decreases to a small number. The average number of c through the whole process, using lin105 as an example, was 12.4. The experimental results in Section IV also show that this modified version of LK typically runs in linear time. Since the local improvement step dominates the computation time, the saved time here directly improves the running time of the whole algorithm.

In addition to restricting the value k by a constant, we apply only one iteration of LK, whereas it is usually applied several times until no more improvement is possible. This also saves considerable CPU time.

D. Computation Time

It is not difficult to check that the time complexity of the crossover operator is $\Theta(n)$. The local improvement takes $O(n^2)$ time in the worst case, but it typically takes only $\Theta(n)$ time in our experiments. All other parts of the GA clearly takes $\Theta(n)$ as is usual in most GAs. Therefore one iteration (generate an offspring, locally improve it, and replace an individual of the population with it) takes $O(n^2)$ time in the-worst case, but typically takes only $\Theta(n)$ time in practice. Since our crossover operator takes linear time, the dominating factor here is the running time of the local improvement step. The modified version of LK, with a typical running time of $O(n)$, helps improve greatly the running time of the entire algorithm.

III. A Preprocessing Heuristic for Problems with Weighted Graphs

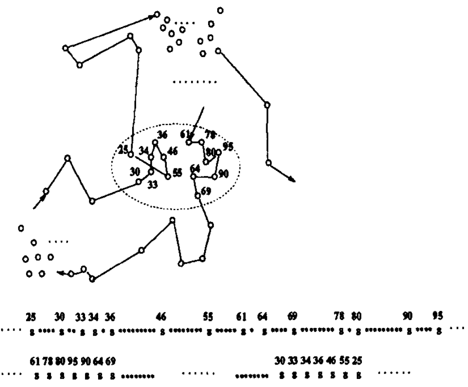
In an instance of TSP a set of indices are given to identify the cities. It is natural to use these indices as the encoding order of cities for the problem. It turned out that the performance of GA can be significantly improved by reindexing the cities. This is useful only when we use a locus-based encoding where each city has a corresponding locus on each chromosome. The idea of reindexing was first suggested by the authors in [2], [3] for unweighted graphs. We showed there that reindexing can significantly improve the performance of GAs for the graph bisection problem. The reindexing heuristics there could not be applied to weighted graphs. Here, we propose a reindexing heuristic for weighted graphs. The fact that we used locus-based encoding is crucial since reindexing technique is of no use to order-based encodings as they don't need any locus assignment for cities. The key idea of reindexing is using clusters in cities' geographical distribution, i.e., a clustered subset of cities has a higher probability of constructing high quality schemata than a set of arbitrary cities. As the reindexing is applied before GAs start, we use the terms reindexing and preprocessing interchangeably.[2] It is apparent that the stronger the clustering, the greater the expected benefit provided by preprocessing.

The given indices of the cities, in order, make a corresponding Hamiltonian path. This Hamiltonian path becomes a Hamiltonian tour by connecting the first city and the last city. We take this tour as an initial tour and improve it by applying our version of LK local optimization but allowing it to have several iterations until no more improvement is made. We then use this order as the encoding order and assign the loci of the cities in the encoding by the reindexed order. It should be noted that this tour has nothing to do with the initial solutions (tours) of the genetic algorithm as the initial population is generated totally at random. In most TSP instances, a geographically clustered subset of cities is likely to have very few visiting patterns in high quality solutions. By this reindexing (locus reassignment), geographically clustered cities tend to have a more clustered distribution on chromosomes than by the given order. Therefore, the loci of those cities are likely to construct high quality schemata.

[2]The term reordering is used in [3], [4].

Consider Figure 3 for an example. We used the symbol 's' to represent the specific positions. The symbol '$*$' invariably represents the non-specific positions. The first schema is from a coding that uses the given indices of cities, i.e., the given input. The second schema is obtained after reindexing the cities. The schemata with corresponding positions of vertices in the dotted circle as defining positions have two clusters defining positions in the reindexed schema and consequently they tend to have a better chance of surviving than the original schema when multi-point crossovers are used. For instance, when a 2-point crossover operator divides a chromosome into three intervals, say intervals 0, 1 and 2, all specific positions of a schema should reside only in interval 1 or only in intervals 0 and 2 for the schema to survive through the operator. We can easily check that the latter schema of Figure 3 has a much higher survival probability than the former in this case. Formal arguments on the advantages of clustered schemata are provided in [4].

Figure 3: Loci reassignment.



IV. Experimental Results

We provide experimental results on some benchmark problems whose optimal tours are known. We compare two versions of GAs for TSP: TSP GA (TGA) and Preprocessed TSP GA (PTGA). Table 1 shows the results gotten through 100 runs of each version on each problem. We used population size 100 and a proportional selection scheme where the best fitted individual has 4 times higher chance to be selected than the worst fitted. We used steady state GA where each time an offspring is generated, it replaces one member of the population instead of performing a group replacement. The most inferior member of the population is replaced by the offspring. The number of crossover points was set to 5. The value k in LK is fixed to 9. In Table

1, the column 'Opt' shows the optimum solutions for the problems. The columns 'Best' and 'Average' show the best and the average in 100 runs of TGA and PTGA, respectively. Figures in parentheses after best and average values represent the percentage above the optimum solution. The column '#LocImp(CPU)' shows the average numbers of local improvements applied before the algorithm finishes and the corresponding CPU seconds are given in parentheses. The algorithm stops when all members of the population converge to the same chromosome. Note that the number of local improvements is the most important metric for the time complexity since it is applied to every offspring generated and is the dominating factor of the algorithm's running time. In a measurement using the tool *gprof*, the local improvement took 96.2% of the total running time on the problem lin105. The other problems also exhibited similar figures.

In general, PTGA produced slightly better results than TGA in *visibly* less time. On the average PTGA takes only 67% of the time taken by TGA on the four lin and kroA graphs. From Table 1, we can get the average time per local improvement for each graph with PTGA as an example, the average CPU seconds per local improvement are 0.061, 0.058, 0.114, and 0.209 for kroA100, lin105, kroA200 and lin318, respectively. This shows some evidence that the typical running time of the local improvement is linear. It should also be noted that the graph lin318 was a very hard graph for the algorithm by Homaifar et al [12] which showed the best result of 42,154 (1.96% above optimum) on lin318. PTGA produced the optimum 41,345 and the average of 100 runs was 41,420.75 (0.18% above optimum), which was also quite better than their best. To get their result, they applied 108,000 $\Theta(n^2)$ local improvement steps while we applied 3,345 $\Theta(n)$ local improvement steps.

Figure 4: Given order of kroA200 and the order after the reindexing.

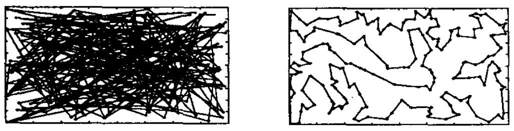


Figure 4 shows the given cities' order and the order after the reindexing. Note that those are not the solutions. The orders are represented by their corresponding Hamiltonian paths. In Table 1, lin graphs didn't show a dramatic running time decrease as kroA graphs with preprocessing. This was not expected since 'lin' graphs seem to be as strongly clustered as 'kroA' graphs. When we carefully examined the given indices of these problems, we found that the indices of 'lin' graphs are more carefully assigned than those of 'kroA' graphs. The given indices of 'lin' graphs happened to be a very favorable order for TGA although they are not comparable to the new order. When we permuted the city order of lin318 graph at random like 'kroA' graphs the performance of TGA on 'lin' graphs became significantly poorer.

V. Conclusions

We presented a new genetic algorithm for solving the TSP with a new encoding/crossover pair and a preprocessing. We believe that this algorithm will also be applicable to most ordering problems with minimum changes except for problem specific variations. The two most important characteristics of this approach are:

1. a new locus-based crossover operator using a linear-sized and locus-based encoding, and
2. improved performance by a static preprocessing step.

As the preprocessing takes less than 1% of the total running time, we can say that it provides visible improvement at negligible cost. From this work and those of [3], [4] and [5], it seems reasonable to claim that the performance of GAs using locus-based encodings is highly dependent on the encoding order of vertices and that the technique of static preprocessing is beneficial to non-hypergraphs, hypergraphs, unweighted graphs and weighted graphs. It should also be mentioned again that the order of the reindexed tour has nothing to do with initial solutions of the population which are generated at random in any case. It only affects the locus assignment of cities in the chromosome encoding. We hope to see the effect of preprocessings on other ordering problems or scheduling problems for which we think the proposed GAs will work well.

References

[1] H. Braun, "On Traveling Salesman Problems by Genetic Algorithms," 1st Workshop on Parallel Problem Solving from Nature, pp. 129–133, Oct., 1990.

[2] T. N. Bui and B. R. Moon, "A Genetic Algorithm for a Special Class of the Quadratic Assignment Problem," to appear in *The Quadratic Assignment and Related Problems*, DIMACS (Center for Discrete Math and Computer Science) Series in Discrete Mathematics and Theoretical Computer Science.

[3] T. N. Bui and B. R. Moon, "Hyperplane Synthesis for Genetic Algorithms," Fifth Interna-

Table 1: Results of TGA and PTGA.

Graphs	Opt	TGA			PTGA		
		Best	Average	#LocImp (CPU[†])	Best	Average	#LocImp (CPU[†])
kroA100	21,282	21,282(0.00[‡])	21,282.23(0.00)	1,288(79)	21,282(0.00)	21,282.00(0.00)	658(40)
lin105	14,379	14,379(0.00)	14,379.22(0.00)	696(40)	14,379(0.00)	14,379.00(0.00)	552(32)
kroA200	29,368	29,368(0.00)	29,375.03(0.02)	3,049(379)	29,368(0.00)	29,373.88(0.02)	1,960(223)
lin318	41,345	41,345(0.00)	41,436.01(0.22)	4,764(903)	41,345(0.00)	41,420.75(0.18)	3,345(698)

[†]CPU seconds on Sun SPARC IPX.
[‡]Values in parentheses under 'Best' and 'Average' are percentages above the known optimal values.

tional Conference on Genetic Algorithms, pp. 102–109, July, 1993.

[4] T. N. Bui and B. R. Moon, "Analyzing Hyperplane Synthesis in Genetic Algorithms Using Clustered Schemata," Technical Report, CS-93-08, Pennsylvania State university, University Park, 1993.

[5] T. N. Bui and B. R. Moon, "A Fast and Stable Hybrid Genetic Algorithm for the Ratio-Cut Partitioning Problem on Hypergraphs," Technical Report, CSE-93-09, Pennsylvania State university, University Park, 1993.

[6] M. Garey and D. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness,* Freeman, San Francisco, 1979.

[7] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning,* Addison-Wesley, 1989.

[8] D. Goldberg and R. Lingle, "Alleles, Loci, and the Traveling Salesman Problem," First International Conference on Genetic Algorithms and Their Applications, pp. 154-159, 1985.

[9] J. Grefenstette, "Incorporating Problem Specific Knowledge into Genetic Algorithms," *Genetic Algorithms and Simulated Annealing,* L. Davis ed., Morgan Kaufmann, pp. 42–60, 1987.

[10] J. Grefenstette, R. Gopal, B. Rosmaita, and D. Gucht, "Genetic Algorithms for the Traveling Salesman Problem," First International Conference on Genetic Algorithms and Their Applications, pp. 160–168, 1985.

[11] J. Holland, *Adaptation in Natural and Artificial Systems,* University of Michigan Press, Ann Arbor, MI, 1975.

[12] A. Homaifar, S. Guan, and G. Liepins, "A New Approach on the Traveling Salesman Problem," Fifth International Conference on Genetic Algorithms, pp. 460–466, July, 1993.

[13] P. Jog, J. Suh and D. Gucht, "The Effect of Population Size, Heuristic Crossover and Local Improvement on a Genetic Algorithm for the Traveling Salesman Problem," Third International Conference on Genetic Algorithms, pp. 110–115, June, 1989.

[14] S. Lin and B. Kernighan, "An Effective Heuristic Algorithms for the Traveling Salesman Problem," Operations Research 21, pp. 498–516, 1973.

[15] E. Lawler, J. Lenstra, A. Kan, and D. Shmoys, *The Traveling Salesman Problem,* Wiley-Interscience Publication, 1985.

[16] I. Oliver, D. Smith, and J. Holland, "A Study of Permutation Crossover Operators on the Traveling Salesman Problem," Second International Conference on Genetic Algorithms, pp. 224–230, July, 1989.

[17] I. Or, *Traveling Salesman-Type Combinatorial Problems and Their Relation to the Logistics of Regional Blood Banking,* Ph. D. Thesis, Northwestern University, Evanston, IL., 1976

[18] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whitley, "A Comparison of Genetic Sequencing Operators," Fourth International Conference on Genetic Algorithms, pp. 69–76, July, 1991.

[19] G. Syswerda, "Uniform Crossover in Genetic Algorithms," International Conference on Genetic Algorithms, pp. 2–9, 1989.

[20] G. Syswerda, "Scheduling Optimization Using Genetic Algorithms," In *Handbook of Genetic Algorithms,* L. Davis ed., Van Nostrand Reinhold, New York, 1990.

[21] N. Ulder, E. Aarts, H. Banbelt, P. Laahoven, and E. Pesch, "Genetic Local Search Algorithms for Traveling Salesman Problem," 1st Workshop on Parallel Problem Solving from Nature, pp. 109–116, Oct., 1990.

[22] D. Whitley, T. Starkweather, and D. Fuquay, "Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator," Third International Conference on Genetic Algorithms, pp. 133–140, 1989.