A Comparison of Operators for Solving Time Dependent Traveling Salesman Problems Using Genetic Algorithms

Leonard J. Testa
Dept. of Comp. Science
North Carolina A&T State University
Greensboro, NC 27411

Albert C. Esterline
Dept. of Comp. Science
North Carolina A&T State University
Greensboro, NC 27411

Gerry V. Dozier
Dept. of Comp. Science and Software Engineering
Auburn University
Auburn, AL 36849

Abdollah Homaifar
Electrical Engineering Dept.
North Carolina A&T State University
Greensboro, NC 27411

Abstract

This paper describes which genetic operators can best solve time dependent traveling salesman problems (TDTSPs) containing up to 50 cities. We first provide an overview of the TDTSP and illustrate its relation to other scheduling and routing problems. Next we describe a genetic algorithm that implements eight common genetic operators, plus Julstrom's adaptive operator probability and Goldberg's population re-initialization mechanisms. We present the results of 280 experiments and show that one combination of these operators and mechanisms outperforms a well-known dynamic programming heuristic. An analysis of the test results indicates that hybrid solutions incorporating solution techniques for both scheduling and the traveling salesman problems may generate better results than either technique alone.

1 INTRODUCTION

The time dependent traveling salesman problem (TDTSP) is a variation of the classic traveling salesman problem (TSP) in which the amount of time it takes the salesman to travel from one city to another fluctuates depending on the time of day the salesman travels. By allowing the travel time between cities to vary, the TDTSP can better model real world conditions such as heavy traffic, road repair, and automobile accidents than can the traditional TSP[1]. Because it incorporates time dependent costs, the TDTSP can also be used to model several well known, fundamental problems in job-shop scheduling and vehicle routing [2][3]. Previous research [4] has shown, however, that genetic operators that work well on the traditional TSP do not work well on tasks such as schedule optimization. Since the TDTSP displays characteristics of both the TSP and scheduling problems, it is unclear which operators would be most effective in generating good solutions to this class of problem.

This paper describes a genetic algorithm (GA) for solving the TDTSP which implements eight well-known genetic operators, plus adaptive operator probabilities and population re-initialization mechanisms, to determine which combination of operators and mechanisms produces the best solutions to a randomly generated TDTSP containing 50 cities. The rest of this paper is as follows. Section 2 provides an overview of the TDTSP, including its relation to job-shop and vehicle routing problems as well as previous research. Section 3 defines the GA in detail, including the operators and mechanisms used. In Section 4 we present the test data and describe how the operators and mechanisms were evaluated. Section 5 discusses the test results and compares the best solutions produced by the GA to that produced by a well-known dynamic programming heuristic. Section 6 presents our conclusions.

2 AN OVERVIEW OF THE TDTSP

The traditional TSP begins with a set of n cities numbered $\{0, 1, \ldots, n\text{-}1\}$ and an $n \times n$ cost matrix D in which the value D_{ij} is the distance from city i to city j [5]. The goal of the TSP is to find the minimal cost tour of the cities in which the salesman visits each city exactly once and returns to the starting city at the end of the tour [6][7]. The TDTSP is a variant of the TSP in which the salesman must still visit each city, but the cost of traveling from city i to city j depends on both the distance matrix D and the time of day the travel takes place [1]. The cost associated with the time of day can be computed by first dividing the day into discrete timeslices of fixed duration, then constructing a cost matrix W, where W_{ijt} is the delay experienced when traveling from city i to city j at time t. Also, when solving the TDTSP, some node (usually node

0) is typically designated the "depot" node, where the salesman begins and ends his tour.

Many real-world instances of the TDTSP are concerned with scheduling time-dependent tasks. Picard and Queryanne [2] described the process of scheduling manufacturing jobs on a machine with time-dependent setup costs. Fox [8] and Fox, Gavish and Graves [9] discussed several single machine scheduling problems as instances of the TDTSP. Another special instance of the TDTSP, known as the deliveryman problem (DMP), was used by Simchi-Levi and Berman [3] to route guided vehicles through a manufacturing system. Other applications of the TDTSP include routing data through a network [10], creating timetables for university exams [11], and scheduling vehicles and crews [12]. Testa, Esterline and Dozier [13] used the TDTSP to model the riding of amusement park attractions where the time spent waiting in line at each attraction varied with the time of day.

As one would expect because the TSP is NP-hard, the TDTSP is also NP-hard [5]. Research indicates, however, that the TDTSP is a more difficult problem. Exact solutions to TSPs involving several thousand cities have been reported [14], but exact solutions to TDTSPs have only been published for problems with a few dozen cities [15]. Typically, these exact solutions also involve restrictions on the kinds of problems solved, such as requiring the travel between cities to be completed in a single time period [5][15]. Several explanations for the relative difficulty of the TDTSP have been put forward, most completely by Malandraki and Daskin [1]. In particular, they note that well-known TSP heuristics, such as the Lin-Kernighan k-opt exchange, are not easily adapted to the TDTSP. When links between nodes are exchanged in the TDTSP, the travel times of many other nodes later in the tour may be affected, and recomputing these travel times can be prohibitively expensive. Malandraki and Daskin also show that certain properties of the optimal solutions to Euclidean TSPs do not extend to the TDTSP. Specifically, they show that the convex hull property present in optimal Euclidean TSP solutions does not hold for the TDTSP. Thus, different heuristics are needed to generate solutions to general TDTSPs with more than a few dozen cities [1].

3 A GENETIC ALGORITHM FOR THE TDTSP

In this section, we describe the construction of the GA in detail. Section 3.1 is an overview of the important features of the algorithm. Section 3.2 describes the various operators that were tested, and Section 3.3 describes the use of adaptive operator probabilities and population reinitialization.

3.1 OVERVIEW

Previous research [7] has shown that GAs alone do not perform as well as those incorporating some sort of local search heuristic. Thus, our GA started by creating an initial solution using a dynamic programming (DP) heuristic first described by Malandraki and Dial in [16]. This effective heuristic avoids the exponential CPU and memory requirements of an exact DP algorithm by retaining in memory only a user-defined number of partial solutions. Retaining more partial solutions generally results in better overall solutions, and storage of tens of thousands of partial solutions to generate good results is not uncommon. In our GA, the DP heuristic was allowed to retain 10 partial solutions in memory at each stage. The result was about 17% greater than the best known path for our test data. This initial solution was added to our GA population, where the chromosomes are permutations of the n-1 integers representing the path of cities to be visited. The remaining members of the population were initialized with random paths.

Binary tournament selection [17] was used to select members of the population for reproduction. Either one or two parents were chosen, depending on the operator selected for that generation. Operator selection was performed randomly where the likelihood of an operator being selected was determined by its associated probability.

During each generation, the decision of whether to retain a new individual o in the population is made using a (P+1) reproduction approach [18]. Specifically, let q be the member of the population with the highest cost tour. The costs of o and q are compared, and if o has a lower cost than q, then q is deleted from the population and o retained.

3.2 OPERATORS

Many operators have been proposed for the TSP and scheduling problems [19]. Several studies [20][4] have indicated that operators that perform well on one of these problems tend not to perform well on the other. The operators chosen for our GA, therefore, have shown to be effective on either the TSP or vehicle routing problems similar to the TDTSP. We also implemented a variety of mutation and local search operators. A brief description of each of the operators used follows.

- **Edge Recombination (ER):** First proposed in [21] for the TSP, edge recombination has also been shown to be effective on certain kinds of scheduling problems [19]. Edge recombination produces a single offspring from two parent paths. The motivating belief behind edge recombination is that the key feature of the TSP is the connections (edges) between cities rather than the positions of the cities in the path, since

it is the connections between cities that represent the cost of travel [22].

Edge recombination works by first building a list of edges present in each parent, then transferring edges found in both parents to the offspring. For each city *k*, the edge list contains the list of cities connected to city *k* in either of the parents.

After the edge list has been constructed, the offspring is constructed by first examining the initial city in each parent path. The initial city that has the smallest number of edges is chosen as the initial city in the offspring. Next, the edge lists of the cities connected to the initial city are examined, and the city with the lowest number of edges is selected as the second city. Subsequent cities are chosen the same way until all cities are present in the path. If at any time the number of edges in two or more cities is equal, one of those cities is chosen at random as the next city to visit. Similarly, if the current city contains no edges, the next city to visit is chosen at random from the remaining unvisited cities.

Our GA implements a modified version of edge recombination in which a greedy heuristic, rather than random selection, is used to resolve ties when recombination becomes blocked. The greedy heuristic estimates the cost of visiting each of the remaining unvisited cities from the current city. The remaining city with the lowest cost becomes the next city to be visited, and edge recombination resumes.

- **Merge Crossover (MX):** Originally proposed in [23] for vehicle routing problems, merge crossover seeks to preserve in one offspring any global precedence of cities found in the offspring's two parents. That is, for any two cities *i* and *j*, if city *i* appears before city *j* in both parents, then city *i* must appear before city *j* in the offspring. Some implementations [24] receive global precedence information from an external source (e.g., a global precedence table). Our implementation, however, seeks to discover global precedents in the parents instead of using an external table. For example, given two parents p_1 = (0 2 1 6 7 3 5 4 8 9) and p_2 = (0 2 1 3 6 5 7 4 8 9), the merge crossover operator first constructs an *after* set for each city in p_1. For each city *i* in p_1, the *after* set of city *i* contains all the cities that appear after *i* in p_1. The *after* set for city 2 in p_1 contains the elements { 1, 6, 7, 3, 5, 4, 8, 9 }, the *after* set for city 1 contains the elements { 6, 7, 3, 5, 4, 8, 9 }, and so on.

When the construction of the *after* set for each city in p_1 is complete, the *after* set for each city in p_2 is created. For each city, the intersection of both *after* sets is then computed, resulting in a global *after* set for that city. The global *after* set for each city is shown below (recall that city 0, the depot node, is always the first node in any path. City 0 is omitted for clarity):

Table 1: Global *After* Set

CITY	***AFTER* SET**
1	{ 6 7 3 5 4 8 9 }
2	{ 1 6 7 3 5 4 8 9 }
3	{ 5 4 8 9 }
4	{ 8 9 }
5	{ 4 8 9 }
6	{ 7 5 4 8 9 }
7	{ 4 8 9 }
8	{ 9 }
9	∅

Once the global *after* set has been constructed, a weight is computed for each city. A city's weight is calculated by counting the number of times the city appears in the *after* set of any other city. For example, city 5 appears in the *after* set of four cities (1, 2, 3, and 6), so the weight of city 5 is 4. The weights of all the cities in our example are shown in the following table:

Table 2: City Weights

CITY	1	2	3	4	5	6	7	8	9
WEIGHT	1	0	2	6	4	2	3	7	8

The cities are placed in the offspring in ascending order according to their weight with ties broken randomly. Thus, our offspring *o* contains the path (0 2 1 3 6 7 5 4 8 9).

- **Cycle Crossover (CX):** Described in [25] for the TSP, cycle crossover produces a single offspring from two parent paths. Cycle crossover is designed to preserve in the offspring the absolute position of each city in the parents. Thus, any city *k* in position *m* in the offspring must also appear in position *m* in one of the parents.

- **Scramble Sublist Mutation (SSM):** First published in [26] and applied to the TSP, scramble sublist mutation produces one offspring from one parent by randomly selecting a sublist of cities from the parent and randomly repositioning the cities within the sublist of the offspring. For example, given a parent p = (0 2 1 | 6 7 3 5 | 4 8 9), and sublist (6 7 3 5) delimited by the symbol |, we rearrange the cities in the sublist at random, then place the scrambled sublist in the same position in the offspring o = (0 2 1 | 3 6 5 7 | 4 8 9). For our implementation the length of the sublist was selected randomly from the range [2, 5].

- **Uniform Order-based Mutation (UOM):** Many GAs for solving the TSP and scheduling problems implement some version of this basic mutation operator [26]. Uniform order-based mutation is a unary operator that works by exchanging in the offspring the positions of two randomly selected cities in the parent. For example, given parent p = (0 2 1 6 7 3 5 4 8 9), we randomly choose cities 6 and 4 to be swapped. This produces the offspring o = (0 2 1 4 7 3 5 6 8 9). Since all valid tours must start at city 0 (the depot), city 0 cannot participate in any of the genetic operators.

- **Non-Uniform Order-based Mutation (NOM):** Similar to uniform order-based mutation, the non-uniform variant also produces one offspring from one parent. In the non-uniform version, however, the average difference between the positions of the two cities to be swapped decreases as the number of generations processed increases. The implementation in our GA began by selecting one city at random. The position of the second city to be swapped was calculated using the following function adapted from [27]:

$$\text{Distance from city 1} = y \cdot r(1 - \frac{t}{T})^b$$

where $y = n - 1$, r is a random number in [0,1], T is the maximal generation number, t is the current generation number, and b is a user-defined parameter used to control the degree of nonuniformity. We set the value of b to 1.4 for our implementation.

- **Uniform Local Search (ULS):** The idea for the uniform local search operator is based on the scramble sublist mutation operator. While scramble sublist randomizes the positions of cities within a sublist of the offspring's path, the uniform local search operator computes the cost of every permutation of the cities in a parent's sublist, then assigns to the offspring the one sublist permutation which minimizes the overall cost of the tour. For example, given a parent

  p = (0 2 1 | 6 7 3 5 | 4 8 9), and sublist (6 7 3 5) of length 4, the uniform local search operator computes the cost of the entire path using each of the 4! permutations of the cities in the sublist. In our example, if the lowest cost overall path is obtained with the permutation (3 5 7 6), then the offspring o would be set to (0 2 1 3 5 7 6 4 8 9).

  While the uniform local search operator is guaranteed to find the lowest cost permutation, the processing time of the operator grows factorially as the length of the sublist grows linearly. Our GA, therefore, used a sublist of length 6, and thus evaluated a total of 720 permutations each time the uniform local search operator was called.

- **Non-Uniform Local Search (NLS):** Unlike uniform local search, the cities selected by the non-uniform local search operator are chosen randomly from the entire path. For example, given a parent p = (0 2 1 6 7 3 5 4 8 9), we select four cities at random, say (2 6 5 8), and mark their positions in the parent with the symbol _ to get p = (0 _ 1 _ 7 3 _ 4 _ 9). Each time a permutation of (2 6 5 8) is computed, we replace one _ symbol in p with one element of the permutation in a left-to-right manner. Thus, if we create the permutation (5 8 6 2), then o would contain the path (0 5 1 8 7 3 6 4 2 9). In this manner, each permutation of the selected cities is created and the corresponding path cost is calculated. The offspring is assigned the path corresponding to the permutation of the selected cities that minimizes the overall cost of the tour. As with the uniform local search operator, non-uniform local search works with just 6 cities to keep the CPU time required by this operator at a reasonable level.

3.3 MECHANISMS USED

In addition to the operators described above, our GA implemented two special-purpose mechanisms to test their effectiveness on the TDTSP: adaptive operator probabilities and population reinitialization. Rather than assigning static probabilities to each genetic operator before runtime, using adaptive operator probabilities allows the GA to adjust the relative probabilities of each operator according to how much relative improvement that operator has contributed to the current population. Several researchers have proposed adaptive operator mechanisms for steady state or generational GAs. Our GA implements the ADOPP (*adaptive operator probabilities*) mechanism

found in [28]. A brief discussion of how ADOPP adjusts each operator's probabilities follows.

For each offspring created that has a cost lower than the median cost of the current population, ADOPP assigns credit to each operator that helped build that offspring, where the amount of credit assigned is a user-defined constant. The operator that directly creates the improved offspring, known as the *immediate* operator, gets the maximum amount of credit, while the operators that generated the offspring's parents get some reduced amount of credit. Credit can be assigned to several generations of ancestors, where the number of ancestor generations and rate of credit decay are user-defined.

Since ADOPP must assign credit to each operator that contributes to a fit offspring, ADOPP keeps track of an *operator tree* for each member of the current population. This operator tree records the operators that generated the individual and its ancestors for a fixed number of prior generations. When a binary operator is applied, for example, ADOPP copies the operator trees of each parent into the left and right subtrees of the offspring's operator tree, discarding each parent's leaf nodes. The current operator becomes the new root. Our GA recorded a maximum of four generations of ancestor operators for each offspring.

In addition to operator trees, ADOPP also maintains a queue that keeps track of each operator's contributions to the population for some user-defined number of most recent previous generations. Let *QLEN* be the length of this queue. For each operator *op* in the GA, the queue contains Cr(*op*), the total amount of credit assigned to *op* over the last *QLEN* generations and N (*op*), the number of individuals in the last *QLEN* generations whose immediate operator was *op*. The contributions of the oldest operator on the queue are removed to make room when a new operator's contributions are added to the queue. The value of *QLEN* was set to 100 for our GA.

After a new individual is added to the population, ADOPP recomputes the probability of selection of each of the *m* operators in the next generation according to the following formula:

$$\text{Probability of operator } op = \frac{Cr[op]}{N[op]} \Big/ \sum_{i=1}^{m} \frac{Cr[i]}{N[i]}$$

To ensure that all operators continued to participate in the GA, the minimum probability of any operator was set to 5%.

In addition to adaptive operator probabilities, our GA also implemented the "population reinitialization" mechanism described in [29]. Population reinitialization is a method of introducing diversity into a population that may have converged prematurely. Reinitialization works by creating a new population where only a few of the best individuals from the old population are copied to the new, and the rest of the new population is created at random. This mechanism has been shown to give good results on problems that use small population sizes [29]. Our implementation of reinitialization copies only the individual with the lowest overall cost from the old population into the new. Reinitialization takes place once 2,500 generations have passed without a new member having been added to the population.

4 TEST DATA AND METHODOLOGY

This section is divided into two parts. Section 4.1 describes how the random test data was generated, and section 4.2 describes the testing methodology.

4.1 TEST DATA

The test data for our 50 city TDTSP was generated randomly and designed to reflect a variety of urban, suburban, and rural driving conditions. The test data consisted of three distinct parts: driving distances between cities, time-dependent traveling costs between cities, and a "service time" at each city. Copies of the test data are available by emailing the corresponding author.

The driving distance between each pair of cities was computed by first placing each city at a unique, random position on an imaginary 50 by 50 plane, then computing the Manhattan distance between each of the cities. Thus, the maximum driving distance between any two cities was 100, and the minimum driving distance was 1. Similarly, each city was assigned a random service time in the range [1,20], with the service time at the depot node set to 0.

The time-dependent traveling costs were also generated randomly within the range [1,50]. Approximately 55% of the traveling costs between cities were designed to model what we believe are typical urban driving conditions. Under these conditions, we assumed that travel delays typically peak three times per day. The first peak occurs during the morning "rush hour", when most commuters are driving to work. The second, smaller peak occurs around noon, when many people go out for lunch. The third peak starts during late afternoon and continues into early evening as commuters drive home. Our urban traveling costs were generated to reflect these peaks.

Approximately 35% of the traveling costs between cities were designed to model typical suburban driving conditions. These costs, while more volatile than rural travel costs, do not experience the fluctuations that their urban counterparts do. We therefore restricted the maximum change between time periods in suburban travel costs to the range [-10,10]. The remaining 10% of the travel costs between cities were considered rural, and held constant throughout the day.

4.2 TESTING METHODOLOGY

We chose to test the eight operators in groups of four. There are 70 unique combinations of four operators chosen from a field of 8. The initial probability of each operator in the 70 combinations was chosen randomly in the range [5,95] so that the sum of all four operators equaled 100%. As demonstrated by the results below, the variation in probabilities of each of the operators had little effect on the success of the tests. Further, each combination was tested with and without adaptive operator probabilities, and with and without population reinitialization, for a total of 280 test combinations. Each test combination was run against the test data 30 times on a 400 MHz. dual Celeron PC with 192 MB of memory and Windows NT Workstation 4.

To compare the quality of solutions generated, we compared each test combination against the solution generated by the Malandraki/Dial dynamic programming heuristic described in section 3.1. The heuristic was allowed to retain 7,500 partial paths and took an average of 65 seconds over 30 runs to generate a solution to the 50-city test problem. The maximum CPU time of the GA was then set to 65 seconds, after which the GA returned the member of the population with the lowest cost. The population size for each test was set to 10.

5 TEST RESULTS AND DISCUSSION

Each test configuration was ranked according to its average best path cost over the 30 trials. The following table shows the three best, three median and three worst test configurations. The mnemonic abbreviations for each operator are listed in section 3.1:

Table 3: Test Results

GA Rank	Operators Used (Initial Probability/Final Probability)	Used Adaptive Prob.?	Used Population Reinit.?	Avg. Cost of Best Path	Std Dev.
1	UOM(48/48), ER(8/8), NOM(36/36), CX(8/8)	No	Yes	1833.1	36.9
	DYNAMIC PROGRAMMING HEURISTIC			1834.0	
2	UOM(19/9), ER(7/63), NOM(10/11), CX(64/17)	Yes	Yes	1852.9	44.3
3	ER(16/16), NLS(11/11), NOM(63/63), CX(10/10)	No	Yes	1858.5	26.7
139	ULS(31/48), MX(15/21), NLS(27/24), CX(27/6)	Yes	No	1900.4	27.2
140	UOM(22/6), ULS(23/56), NLS(33/32), NOM(22/6)	Yes	No	1900.8	26.3
141	ULS(24/48), NLS(23/24), NOM(40/5), CX(13/23)	Yes	No	1901.3	27.3
278	ULS(39/45), SSM(30/13), ER(18/20), CX(13/22)	Yes	No	1994.7	36.4
279	SSM(16/32), ER(49/25), MX(24/21), CX(11/22)	Yes	No	2033.9	3.4
280	SSM(17/17), ER(36/36), MX(23/23), CX(24/24)	No	No	2034.5	4.4

The three best test combinations all include edge recombination, cycle crossover, and at least one of the mutation operators. Notably, the best test configuration was able to generate solutions consistently better than that

of the dynamic programming heuristic. In fact, edge recombination was present in each of the twenty best-performing test configurations, with cycle crossover appearing in 60% and each mutation operator appearing in 55%. One of the two forms of mutation appeared in 70% of the twenty best test configurations.

The median test combinations appear to use non-uniform local search in place of edge recombination and cycle crossover, but still retain mutation. Non-uniform local search is present in 90% of the median twenty test combinations. The operator that appears next most frequently (65%) in the median is uniform local search, followed by scramble sublist mutation (60%). Edge recombination (30%) and cycle crossover (20%) are the least represented operators.

It is also significant that all three of the worst test combinations make use of both edge recombination and cycle crossover but use neither mutation nor local search. The absence of mutation, rather than the presence of other operators is the striking feature of the worst-performing test combinations. In fact, both mutation operators are absent from 95% of the twenty worst test combinations, although uniform local search (85%), merge crossover (85%), cycle crossover (80%), and edge recombination (70%) are all well-represented.

That the three best test combinations all make use of edge recombination, cycle crossover, and at least one of the mutation operators confirms the findings in [4] on the relative effectiveness of edge recombination over other kinds of crossover for certain kinds of scheduling problems. This research also noted that cycle crossover performed moderately well on the TSP. Our results extend these previous findings by illustrating the symbiotic relationship between these operators and relatively high levels of mutation. Where mutation and edge recombination are not present in a test configuration the GA is able to compensate somewhat by the use of local search operators, although solution quality is diminished. Where edge recombination and cycle crossover are not accompanied by mutation, solution quality is poor.

The second observation that can be made from our results is the importance of population reinitialization when the GA has converged. Seventeen of the twenty best test configurations (85%) used population reinitialization. Population reinitialization, therefore, may be considered a requirement for the TDTSP. Conversely, only 7 of the top 20 results used adaptive operator probabilities (35%). While we maintain our belief in the general effectiveness of adaptive operator probabilities, more research needs to be done to utilize it effectively in this class of problem.

The final observation made is that the TDTSP seems to respond well to a mixture of scheduling and TSP techniques. This is demonstrated by the effectiveness of both the edge recombination and cycle crossover operators. Where edge recombination has been shown to perform well on scheduling problems, cycle crossover has performed poorly [4]. This study, however, showed cycle crossover to be moderately effective on the TSP. Researchers investigating TDTSP-like problems might be better off adopting hybrid techniques used successfully in other kinds of scheduling problems and the TSP.

6 CONCLUSIONS

This paper began by describing the time dependent traveling salesman problem and its relation to several problems in the areas of scheduling and vehicle routing. We then described a genetic algorithm for the TDTSP that used eight well-known genetic operators plus adaptive operator probabilities and population reinitialization. Our results indicate that edge recombination, cycle crossover, and relatively high levels of mutation are likely to generate solutions with good results. In particular, we showed that one such combination of these operators, along with population reinitialization, generated higher quality solutions than a previously published dynamic programming heuristic given the same amount of CPU time. We then showed that these crossover operators without mutation performed poorly on our test data, indicating that mutation, even at relatively high levels, is an integral part of a GA solution. We concluded that the application of both scheduling and TSP problem-solving genetic operators and heuristics show promise for generating good solutions to the TDTSP and related problems.

Acknowledgments

The authors wish to thank Jim Kane of the Automation Technologies department of American Express Travel Related Services, Greensboro, NC.

References

[1] Malandraki, C., and Daskin, M., "Time Dependent Vehicle Routing Problems: Formulations, Properties and Heuristic Algorithms", *Transportation Science*, Vol. 26, No. 3 (1992) 185-200.

[2]Picard, J.C., and Queyranne, M., "The Time-Dependent Traveling Salesman Problem and its Application to the Tardiness Problem in One Machine Scheduling" *Operations Research*, Vol. 26 (1978), 86-110.

[3] Simchi-Levi, D., and Berman o., "Minimizing the Total Flow Time of n Jobs on a Network", *IIE Transactions* 23 (1991), 236-244.

[4] Starkweather, T., McDaniel, S., Mathias, K., Whitley, C., and Whitley, D., "A Comparison of Genetic

Sequencing Operators" in *Proceedings of the Fourth International Conference on Genetic Algorithms,* Morgan Kaufmann Publishers, San Mateo, CA, (1991), 69-76.

[5] Vander Wiel, R., and Sahinidis, N., "Heuristic Bounds and Test Problem Generation for the Time-Dependent Traveling Salesman Problem", *Transportation Science,* Vol. 29, No. 2 (1995), 167-183.

[6] Freisleben, B. and Merz, P., "A Genetic Local Search Algorithm for Solving Symmetric and Asymmetric Traveling Salesman Problems", *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation,* (1996), 616-621.

[7] Tamaki, H., Kita, H., Shimizu, N., Maekawa, K., and Nishikawa, Y, "A Comparison Study of Genetic Codings for the Traveling Salesman Problem", IEEE International Conference on Evolutionary Computing, 1994, 2-3.

[8] Fox, K., "Production Scheduling on Parallel Lines with Dependencies", Ph.D. Thesis, Johns Hopkins University, Baltimore, MD, 1973.

[9] Fox, K., Gavish, B., and Graves, S., "The Time Dependent Traveling Salesman Problem and Extensions", Working paper No. 7904, Graduate School of Management, University of Rochester, NY, 1979.

[10] Orda, A., and Rom, R., "Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length", *Journal of the ACM* 37 (1990), 607-625.

[11] Balakrishnan, N., Lucena, A., and Wong, R.T., "Scheduling Examinations to Reduce Second-Order Conflicts", *Computers in Operations Research*, Vol. 19 (1992), 353-361.

[12] Bodin, L., Golden, B., Assad A., and Ball, M., "The State of the Art in the Routing and Scheduling of Vehicles and Crews", Office of Policy Research, Urban Mass Transportation Administration, U.S. Department of Transportation, Report UTMA/BMGT/MSS#81-001, Washington DC., 1981.

[13] Testa, L., Esterline, A., and Dozier, G., "Evolving Efficient Theme Park Tours", *Journal of Computing and Information Technology*, Vol. 7, No. 1 (1999), 77-92.

[14] Ruland, K., "Polyhedral Solution to the Pickup and Delivery Problem", Ph.D. Thesis, Washington University, Saint Louis, MO, 1995.

[15] Vander Wiel R., and Sahinidis, N., "An Exact Solution Approach for the Time-Dependent Traveling Salesman Problem", *Naval Research Logistics*, Vol. 43 (1996), 797-820.

[16] Malandraki, C., and Dial, R.B., "A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem", *European Journal of Operational Research* 90 (1996), 45-55.

[17] Goldberg, D. E., and Deb, K., "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms", Foundations of Genetic Algorithms (FOGA-1), Morgan Kaufman, 1991, 69-93.

[18] Baeck, T., Hoffmeister, F., and Schwefel, H.P., "A Survey of Evolution Strategies", Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann, 1991, 2-9.

[19] Michalewicz, Z., "Genetic Algorithms + Data Structures = Evolution Programs", 3rd Edition, Springer Verlag, 1996, 242-243.

[20] Syswerda, G., "Schedule Optimization Using Genetic Algorithms", Handbook of Genetic Algorithms, Van Nostrand Reinhold, 1991, 332-349.

[21] Whitley, D., Starkweather, T., and Fuqya, D'A., "Scheduling Problem and Traveling Salesman: The Genetic Edge Recombination Operator", *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, 1989, 133-140.

[22] Homaifar, A., and Guan, S., "A New Approach on the Traveling Salesman Problem by Genetic Algorithm", Technical Report, North Carolina A&T State University, 1991.

[23] Blanton (Jr), J.L., and Wainwright, R.L., "Multiple vehicle routing with time and capacity constraints using genetic algorithms", *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, 452-459.

[24] Louis, S., Yin, X., and Yuan, Z., "Multiple Vehicle Routing With Time Windows Using Genetic Algorithm", *Proceedings of the 1999 Congress on Evolutionary Computing,* 1804-1808.

[25] Oliver, I.M., Smith, D.J., and Holland, J.R.C., "A Study of Permutation Crossover Operators on the Traveling Salesman Problem", *Proceedings of the Second International Conference on Genetic Algorithms,* Lawrence Erlbaum Associates, 1987, 224-230.

[26] Davis, L., (Editor), Handbook of Genetic Algorithms, Von Nostrand Reinhold, 1991.

[27] Eiber, A.E., Hinterding, R., and Michalewicz, Z., "Parameter Control in Evolutionary Algorithms", *IEEE Transactions on Evolutionary Computation*, Vol. 3, No. 2, 1999, 124-141.

[28] Julstrom, B., "What Have You Done For Me Lately? Adapting Operator Probabilities in a Steady-State Genetic Algorithm", In *Genetic Algorithms: Proceedings of the Sixth International Conference* (ICGA'95) (Larry J. Eshelman, Ed.). San Mateo, CA: Morgan Kaufmann Publishers, 81-87.

[29] Michalewicz, Z., "Genetic Algorithms + Data Structures = Evolution Programs", 3rd Edition, Springer Verlag, 1996, 174.