Implementation of an Effective Hybrid GA for Large-Scale Traveling Salesman Problems

Hung Dinh Nguyen, *Member, IEEE*, Ikuo Yoshihara, *Member, IEEE*, Kunihito Yamamori, *Member, IEEE*, and Moritoshi Yasunaga, *Member, IEEE*

***Abstract*—This correspondence describes a hybrid genetic algorithm (GA) to find high-quality solutions for the traveling salesman problem (TSP). The proposed method is based on a parallel implementation of a multipopulation steady-state GA involving local search heuristics. It uses a variant of the maximal preservative crossover and the double-bridge move mutation. An effective implementation of the Lin–Kernighan heuristic (LK) is incorporated into the method to compensate for the GA's lack of local search ability. The method is validated by comparing it with the LK–Helsgaun method (LKH), which is one of the most effective methods for the TSP. Experimental results with benchmarks having up to 316 228 cities show that the proposed method works more effectively and efficiently than LKH when solving large-scale problems. Finally, the method is used together with the implementation of the iterated LK to find a new best tour (as of June 2, 2003) for a 1 904 711-city TSP challenge.**

***Index Terms*—Hybrid genetic algorithm, maximal preservative crossover (MPX), memetic algorithm, traveling salesman problem (TSP).**

I. Introduction

The traveling salesman problem (TSP) is one of the most important and representative combinatorial optimization problems because it is simple to state but difficult to solve. The TSP can be stated as follows. The salesman must visit a list of cities, all of whose locations are given. The salesman's task is to find the cheapest tour connecting them all, visiting each city only once, and return to the city of origin. Cost here can be distance, time, money, etc. If all the costs between any two cities are equal in both directions, the problem is called symmetric TSP; otherwise, it is called asymmetric. This correspondence deals only with symmetric TSPs.

Since TSP is NP-complete, exact algorithms (e.g., [1]) are only applicable to TSP instances having several thousand cities. In order to tackle larger instances, it is necessary to develop approximate algorithms that do not always aim at finding optimal solutions but at finding quasi-optimal solutions in an acceptable running time. Johnson and McGeoch [19], [20] provide an excellent survey on approximate algorithms for the TSP. These methods constitute a broad range of time–quality tradeoffs. Tour construction heuristics are fast but find tours of moderate quality. Tour improvement heuristics such as 2-Opt, 3-Opt, and the Lin–Kernighan heuristic (LK) [23] produce better tour quality but need more running time. Iterated local search frameworks [2], [24] are utilized when higher tour quality is needed and more running time is available.

Manuscript received May 1, 2005; revised January 16, 2006, April 30, 2006, and May 10, 2006. This paper was recommended by Guest Editor Y. S. Ong.

H. D. Nguyen was with the Graduate School of Engineering, University of Miyazaki, Miyazaki 889-2192, Japan. He is now with the Frontier Science Research Center, University of Miyazaki, Miyazaki 889-1692, Japan (e-mail: ndhung@med.miyazaki-u.ac.jp).

I. Yoshihara and K. Yamamori are with the Faculty of Engineering, University of Miyazaki, Miyazaki 889-2192, Japan (e-mail: yoshiha@cs.miyazaki-u.ac.jp; yamamori@cs.miyazaki-u.ac.jp).

M. Yasunaga is with the Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba 305-8573, Japan (e-mail: yasunaga@is.tsukuba-u.ac.jp).

Digital Object Identifier 10.1109/TSMCB.2006.880136

General search methods such as genetic algorithms (GAs) [10], [17] have also been applied to the TSP. This is because GAs are global search algorithms appropriate for problems with huge search spaces such as the TSP. The earliest attempts at applying GAs to the TSP include the works of Goldberg and Lingle [9], Grefenstette *et al.* [12], and Whitley *et al.* [41], [42]. They followed the strategy of using pure GAs and focused on developing appropriate chromosome encodings and genetic operators. Their methods, however, were tested only on small-scale TSPs, and the results were rather discouraging when compared with local searches. The methods spend much more time and find worse tours than does LK.

GAs, however, can be used in combination with local search heuristics to produce very high quality solutions. A hybrid GA [also-called a memetic algorithm or a genetic local search (GLS)] can combine the global search ability of a GA with the local search ability of heuristics, potentially being a more powerful search algorithm than either. The most important things when designing a competent hybrid GA are the choice of GA model, the way of incorporating local search into the GA, and the balance between global and local searches [18], [22], [35]. These seem related to each other and may depend on the problem we want to solve. Many hybrid GAs have been proposed for the TSP. Among the best of these methods are the compact GA with LK local search (Cga-LK) [3], the asynchronous parallel genetic optimization strategy (Asparagos) [11], the GA with natural crossover (NGA) [21], the GLS [26], [27], the GA with edge assembly crossover (GA-EAX) [30], and the LK–Helsgaun method (LKH) [15]. These methods are quite successful for solving instances having up to several thousand cities. Only GLS and LKH have been tested for large-scale instances with up to 100 000 cities.

Asparagos relies on a powerful hierarchy multipopulation GA but uses a rather weak "two-repaired" local search for improving the initial population and each offspring. In contrast, GLS and NGA use the more powerful LK local search but only with small population-sized GAs. GA-EAX employs a GA with moderate population size and incorporates local search directly into the EAX crossover. Both Cga-LK and LKH use GAs with an implicit population, i.e., only the global information of the population is stored either by an edge probability matrix or an edge pool. They both employ LK for the local search. However, Cga-LK uses the standard iterated LK with the number of iterations increasing with the number of generations, while LKH uses a modified LK that can search for a significantly larger neighborhood.

Helsgaun does not classify LKH as a hybrid GA [15]; in our view, however, it is. The algorithm starts by building an edge pool. For each iteration, a tour construction heuristic is used to construct an initial tour using information from the current best tour and the edge pool. Then, a modified LK, which uses 5-Opt basic moves and searches for a limited number of nonsequential moves, is used to improve the initial tour. If the tour has a shorter length than the current best tour, its edge will be used to update the edge pool. Thus, the algorithm can be viewed as a hybrid GA where the population is hidden by using the edge pool, the tour construction heuristic plays the role of the crossover and/or mutation operator, and the modified LK plays the role of local search. LKH appears to be the most effective heuristic for the TSP (in terms of solution quality) proposed to date. The drawback of LKH is its scalability. Its running time grows approximately as $O(N^{2.2})$ (throughout this correspondence, N is used to denote the number of cities). This prevents LKH from being applied directly to instances having 100 000 or more cities.

The objective of our research is to develop a more effective hybrid GA capable of finding high-quality solutions for large-scale TSPs (i.e.,

several hundred thousand cities). A previous version of our method has been published elsewhere [32]. Our method uses the greedy subtour crossover (GSX) and double-bridge move mutation. Two kinds of heuristics are utilized, namely: 1) random insertion heuristic for creating the initial population of a GA, and 2) LK for improving offspring produced by crossover or mutation.

Several enhancements are proposed in the present version. First, the GSX is replaced by a variant of the maximal preservative crossover (MPX). Second, faster construction heuristics are used for generating the initial population of a GA for Euclidean and geographical TSPs. Third, some recent improvements in LK, including 5-Opt basic moves [15] and the two-level tree segment lists for tour representation [33], are incorporated into our method. Fourth, an iterated LK with a small number of iterations utilizing random-walk kicks [2] is used for improving the mutations' offspring. Fifth, before applying LK to crossovers' offspring, all common subtours having lengths longer than the root square of the problem size are locked to prevent them from being broken by the tour improvement heuristic. Finally, the method is extended to parallel execution to take full advantage of the GA's inherent parallelism.

II. Hybrid GA for TSP

A. General Flow of Hybrid GA

Our hybrid GA for the TSP is expressed by the pseudocode in Fig. 1. In this pseudocode, the terms in italics (nine in total) are variable parameters that can be controlled, and the terms beginning with capital letters are functions.

Our method is based on a multipopulation GENITOR-type GA. GENITOR was originally proposed by Whitley [40]. It has some distinguishing characteristics, namely: 1) using a steady-state update strategy; 2) using linear ranking for selecting parents; and 3) replacing the worst individual in the population. In our method, however, the offspring replaces the worst parent instead of the worst individual. This replacement scheme causes slower convergence but maintains a more diverse population for a GA.

The population contains "num_subpop" subpopulations, each having "subpop_size" individuals. At the initialization step, each individual is first generated by the tour construction heuristic (function `Construct_tour`) and then improved by the tour improvement heuristic (`Improve_tour`). In every generation and for each subpopulation, only one offspring is produced, either from one parent by mutation (`Mutate`) or from two parents by crossover (`Crossover`), and followed by the tour improvement heuristic. The rate of mutation is decided by variable "mutation_rate." Parents are selected from the population by using linear ranking selection (`Linear_select`) with parameter "selection_bias." If the offspring is better than the worst parent, it is immediately inserted into the subpopulation to replace (`Replace`) the worst parent. However, duplicate individuals are not allowed in each subpopulation. The algorithm halts if either the best individual has not been improved for a predefined number of successive generations ("max_nonimproved_gen") or the total number of generations has reached a predefined limitation ("max_gen"). Several best individuals ("num_migrant") are exchanged (`Migrate`) between subpopulations at predefined migration intervals ("migration_interval"). Our method follows the same migration topology as GENITOR [40].

Since our method is based on a multipopulation GA, we apply the so-called coarse-grained parallel model [5] to the parallel implementation of our method. In this model, each subpopulation is assigned to a separate GA that is run by either a process in a multiprocessor machine or a PC in a PC cluster. Thus, the parallel implementation of our multipopulation hybrid GA is equivalent to the sequential implementation, but it can run on either a multiprocessor machine or a PC cluster. The two implementations will produce the same tour if the same population size, number of subpopulations, and random seed are used. Hereafter, we will use SHGA and PHGA to denote the sequential and parallel implementations of our method, respectively.

```
program HybridGA
begin
 for (sub = 1; sub <= num_subpop; sub++) {
  for (ind = 1; ind <= subpop_size; ind++) {
   Construct_tour(P[sub][ind]);
   Improve_tour(P[sub][ind]);
  }
 }
 best_so_far = Find_best(P);
 num_nonimproved_gen = 0;
 for (gen = 1; gen <= max_gen; gen++) {
  for (sub = 1; sub <= num_subpop; sub++) {
   if (Rand() < mutation_rate) {
    p1 = Linear_select(P[sub], selection_bias);
    c = p1;
    for (iter = 1; iter <= num_iteration; iter++) {
     cc = Mutate(c);
     Improve_tour(cc);
     if (cc.cost < c.cost)
       c = cc;
    }
   } else {
    p1 = Linear_select(P[sub], selection_bias);
    p2 = Linear_select(P[sub], selection_bias);
    c = Crossover(p1, p2);
    Lock_comon_subtours(p1, p2);
    Improve_tour(c);
    Unlock_common_subtour();
    if (p1.cost < p2.cost)
      Swap(p1, p2);
   }
   if (c.cost < p1.cost)
    Replace(c, p1, P[sub]);
  }
  best = Find_best(P);
  if (best < best_so_far) {
   best_so_far = best;
   num_nonimproved_gen++;
  } else {
   if (++num_nonimproved_gen >= max_nonimproved_gen)
    break;
  }
  if ((gen % migration_interval) == 0)
   Migrate(P, num_migrant);
 }
 Report(best_so_far);
end
```

Fig. 1. Pseudocode of the hybrid GA for solving TSP.

B. Local Search Heuristics

Our method uses two kinds of heuristics, namely: 1) tour construction heuristics for constructing the initial population and 2) tour improvement heuristics for improving the initial population and each offspring produced by mutation and crossover.

1) Heuristics for Tour Construction: Different tour construction heuristics are used for different distance types as follows: quick Boruvka for (pseudo-)Euclidean TSPs, nearest neighbor for geographical TSPs, and random insertion for matrix TSPs. These heuristics are described in detail in [20]. Our implementations of the quick Boruvka and nearest neighbor heuristics are based on those of Applegate *et al.* [2].

2) Heuristics for Tour Improvement: Like many other hybrid GAs, our method also uses LK for tour improvement. LK is widely recognized as one of the most successful local search heuristics for the TSP. Our LK implementation uses 12-quadrant nearest neighbors for

Euclidean distance problems and 12 nearest neighbors for others. The LK search is limited to a maximum depth of 100 edges. Besides some standard improvement techniques such as "don't-look bits" and distance caching [4], we also incorporate into our LK implementation two newly proposed improvements, namely: 1) Helsgaun's 5-Opt basic moves [15] and 2) "two-level tree segment list" data structure [33]. The former makes LK more powerful while the latter makes it twofold faster than the conventional two-level tree data structure [7] without any loss in tour quality.

C. Genetic Operators

1) Crossover: It is commonly recognized in the GA community that crossover is the most important operator in GAs. For this reason, many crossovers have been proposed for the TSP, including partially mapped crossover (PMX) [9], cycle crossover [34], MPX [11], [28], [33], edge recombination (ERX) [6], [25], [31], [38], [41], EAX [29], [30], distance preserving crossover (DPX) [8], [26], generic crossover (GX) [27], NGA [21], and GSX [32], [37], [39]. These crossovers can be divided into two classes. Those in the first class use local problem-dependent information (e.g., edge lengths) in generating offspring, whereas crossovers in the second class do not (although the latter can use some global information, e.g., the fitness of a TSP chromosome relies on edges between loci and not the loci themselves). Variants of EAX, DPX, and GX belong to the first class, and variants of MPX, ERX, and GSX belong to the second class.

Our strategy is not to use local information in crossovers. If more local search power is needed, it is better to incorporate it directly into the local search heuristic. In a previous work [32], we used GSX2, an improved variant of the GSX operator, for our method. This operator has the advantage of quickly exploiting the diversity of the GA population that causes the method to converge very quickly. It works very well for problems with small-scale and medium-scale TSPs having up to several thousands of cities. However, for larger-scale TSPs, it often causes the hybrid GA to become trapped in the local minima. In order to find a more appropriate crossover for large-scale TSPs, we therefore compare GSX2 with two other alternatives: the variant ERX6 [31] of the ERX operator and the variant MPX3 [33] of the MPX operator (see Section III-B).

After each crossover, we apply the tour improvement heuristic to the offspring. In order to focus the search, the function `Lock_common_subtours(p1, p2)` (see Fig. 1) is called before invoking the heuristic to lock all common subtours of the two parents that have more than $N^{1/2}$ edges. This is so the heuristic will not break these subtours when trying to improve the offspring. The function `Unlock_common_subtours()` unlocks these subtours after calling the tour improvement heuristic so that they will not affect the heuristic later. By doing this, the heuristic runs much faster, and the offspring still inherits important genetic information from parents.

2) Mutation: In our previous version [32], a single call to the mutation operator followed by a call to the tour improvement heuristic was applied to each individual being mutated. In this version, however, a number "num_iteration" of iterations are applied. For each iteration, the mutation operator performs on the individual a special kind of nonsequential four-exchange move, i.e., the double-bridge move. The tour improvement heuristic is then applied. If the new tour produced by the tour improvement heuristic is not better than before mutation, it is discarded. Otherwise, the old tour is replaced by the new one.

The double-bridge move acts as a kick to alter the current tour before applying the tour improvement heuristic. It has been proven to be very effective when working in the framework of iterated local searches with 2-Opt, 3-Opt, and LK [24]. This is because the move cannot be easily undone by these heuristics. Various methods have been proposed for selecting the edges of the move. Our method applies the random-walk move because this move has been proven to be more efficient than the commonly used random move [2]. However, based on a limited number of experiments (data not shown), we decided to use the value $k = 250$ (k is the number of steps in the random walk) in our hybrid GA since it produced slightly better results than the suggested value $k = 50$ in the framework of iterated LK.

TABLE I
PHGA PARAMETERS USED FOR THE EXPERIMENTS

Parameter	Value	Parameter	Value
num_subpop	10	*num_iteration*	5
subpop_size	50	*max_nonimproved_gen*	2,000
max_gen	500,000	*migration_interval*	500
mutation_rate	0.1	*num_migrant*	3
selection_bias	1.25		

III. EXPERIMENTS AND VALIDATIONS

A. Experimental Settings

The parameters for PHGA were set empirically as shown in Table I. PHGA was performed on a cluster of ten PCs (933-MHz Pentium III, 128-MB memory, free BSD 4.7). The code was written in C and compiled using GNU `gcc` compiler (version 2.95) with the "−O2" optimization option. Executables of our method are available on request to the first author.

As mentioned in Section I, of the hybrid GAs that have been proposed for the TSP, only GLS and LKH have been tested for large-scale TSPs. However, LKH produced much better tour quality than GLS. Therefore, in this section, LKH will be used as the benchmark to validate the effectiveness of our method. We used the file `LKH.UNIX`, which is the executable of LKH code (version 1.3) for the UNIX operating system [15]. All parameters (except the number of runs) were set as default. The LKH method was run on one of the cluster PCs mentioned above.

TSP benchmarks were taken from three sources, namely: 1) TSPLIB benchmark suite [36]; 2) testbed that has been used in [20] (http://www.research.att.com/dsj/chtsp); and 3) World TSP Challenge (http://www.tsp.gatech.edu/world).

B. Contributions of New Enhancements

We carried out the first set of experiments to assess the contributions of the improved LK heuristic and the two crossovers ERX6 and MPX3. Four TSP instances, ranging in size from 3038 to 15 112 cites in TSPLIB [36], were used for this purpose. (For all TSPLIB instances, the suffix numbers show the size of instances.) For each instance, 30 runs were performed, and the results are given in Table II.

In this table, the column "Name (optimal)" shows the instance name and the optimal value in parentheses. The column "Crossover/Local search" displays the combinations of crossover and local search; LK is the standard LK heuristic with 2-Opt basic moves, and LK5Opt is the improved LK heuristic with 5-Opt basic moves. The columns "Best" and "Average" give the best and average tour lengths, respectively, of 30 runs. Finally, the column "Times (s)" indicates the average running time of SHGA in seconds.

For the first two combinations that use GSX2, the tour quality of the method was improved for all instances when LK5Opt was used instead of LK. T-tests show that all of these improvements were statistically significant ($P < 0.05$). However, the method was two to three times slower. Among the three combinations that employ LK5Opt, GSX2 was the fastest crossover but produced the worse tour quality. The

TABLE II
COMPARISON OF LOCAL SEARCH HEURISTICS AND CROSSOVERS

Name (optimal)	Crossover/ Local search	Best	Average	Time (s)
	GSX2/LK	Optimal	137,703.6	**149.1**
pcb3038	GSX2/LK5Opt	Optimal	137,694.8	443.8
(137,694)	ERX6/LK5Opt	Optimal	137,694.3	920.6
	MPX3/LK5Opt	Optimal	**Optimal**	889.7
	GSX2/LK	23,260,814	23,264,086.7	**500.9**
pla7397	GSX2/LK5Opt	**Optimal**	23,261,017.7	1,478.8
(23,260,728)	ERX6/LK5Opt	23,260,814	23,261,416.0	67,594.8
	MPX3/LK5Opt	**Optimal**	**23,260,805.4**	3,816.7
	GSX2/LK	923,503	923,838.0	**730.8**
rl11849	GSX2/LK5Opt	923,318	923,562.8	2,206.5
(923,288)	ERX6/LK5Opt	923,303	923,387.5	20,030.9
	MPX3/LK5Opt	**Optimal**	**923,338.9**	5,984.2
	GSX2/LK	1,574,025	1,574,294.8	**1,941.0**
d15112	GSX2/LK5Opt	1,573,300	1,573,611.3	3,575.2
(1,573,084)	ERX6/LK5Opt	1,573,454	1,573,572.8	73,826.4
	MPX3/LK5Opt	**1,573,183**	**1,573,340.5**	15,392.0

SHGA was used. Times were measured on a 3.0-GHz Xeon PC.

TABLE III
SPEEDUPS OF PHGA OVER SHGA FOR TSPLIB INSTANCES

Name	SHGA Time (s)	PHGA Time (s)	Speedup
pr1002	529.68	70.05	7.56
pcb1173	418.22	55.47	7.54
rl1304	669.64	87.57	7.65
nrw1379	945.15	120.25	7.86
pr2392	948.17	117.30	8.08
pcb3038	2,463.11	358.65	6.87
fnl4461	2,709.63	364.92	7.43
pla7397	8,872.21	1,141.31	7.77
brd14051	33,719.93	4,698.51	7.18
pla33810	150,877.48	21,250.33	7.10
pla85900	343,567.95	42,802.40	8.03

SHGA was run on a 933-MHz Pentium III PC; PHGA on a cluster of ten 933-MHz Pentium III PCs.

ERX6 operator had slightly better tour quality than GSX2 but suffered from a very slow speed. MPX3 was two to four times slower than GSX2 but obtained the best average tour quality for all four instances. All of the improvements in tour quality by replacing GSX2 with MPX3 were statistically significant ($P < 0.05$, t-tests). Since tour quality was more important for us, we chose LK5Opt as the local search and MPX3 as the crossover.

We performed another set of experiments to measure the speedups of PHGA over SHGA for the 11 instances in TSPLIB (Table III). In this table, the columns "SHGA Time" and "PHGA Time" show the running times in seconds of the sequential and parallel implementations of our method for a single run, respectively. The column "Speedup" shows the speedups of PHGA over SHGA for each instance. The table shows that the speedups vary from seven to eight times.

C. Comparison of PHGA and LKH for TSPLIB Instances

We compared our PHGA with LKH using all 34 instances having from 1000 to 85 900 cities in TSPLIB. For each instance, ten runs were performed. However, because LKH may need an enormous amount of CPU time to solve the largest instance `pla85900`, only one run of LKH $-$ $.1N$ (i.e., LKH with $N/10$ iterations) was performed for this instance. For both PHGA and LKH, the same parameters were used across all instances. Table IV shows the comparative results of PHGA and LKH for these 34 instances.

In this table, the column "Name" indicates the instance name in TSPLIB; the columns "Best," "Average," and "Worst" show the best, average, and worst tour lengths of ten runs, respectively (the values in parentheses are the percentage above the optimal/lower bound); the column "St. Dev." gives the standard deviation of the ten tour lengths; the column "Gen." displays the average number of generations; the column $N_{\rm opt}$ shows the number of times an optimal solution was found; the column "Time (s)" indicates the average running time in seconds; and the column "P (t-test)" shows the P values for t-tests comparing the average tour qualities of the two methods. In order to make a fair comparison, the running times of PHGA were normalized to a single machine based on the speedups in Table III. For each instance x in Table IV, the speedup of the next largest instance in Table III was used. For example, the speedup of the instance `pla7397` was used to normalize the running times of all instances having sizes in the range $4461 < N \leq 7397$.

Helsgaun [15] reported that, except for two instances `rl5915` and `rl5934`, LKH was able to find optimal tours in at least one out of ten runs for all instances having less than 10 000 cities. In our experiments, however, LKH failed to find optimal tours for four other instances in this size range, namely: 1) `fl1400`; 2) `fl1577`; 3) `u1817`; and 4) `d2103`. Furthermore, LKH found an optimal tour for `fl3795` in only one out of ten runs, while PHGA found an optimal tour for this instance in all runs. Most of these instances have been known as heavily clustered, so LKH appears to perform badly on that class of TSP.

In terms of tour quality, PHGA found better average tour qualities than did LKH for 26 out of 34 instances. Improvements in 15 of these 26 instances were statistically significant at the $P < 0.05$ level. PHGA produced worse average tour qualities for only three instances, namely: 1) `u2319` ($P < 0.05$); 2) `brd14051` ($P < 0.05$); and 3) `d15112` ($P > 0.05$). It produced the same tour qualities for the remaining five instances. Ideally, more runs should be performed to increase the significance of the tests. However, this would have taken too much time to do.

In terms of running time, PHGA was slower than LKH on TSPs having up to 5934 cities. The reason is partly because the time PHGA spent for checking the stop condition accounted for a large portion of the total running time. The condition for PHGA to stop is that the best tour in the population has not been improved for 2000 successive generations. For most of these instances, PHGA converged at an average of less than 4000 generations. Therefore, it is possible to reduce the PHGA time for these instances by using more reliable criteria for the stop condition. We decided to leave this for future work since the main target of our method is large-scale TSPs. For all larger TSPs, however, PHGA was faster than LKH. The ratio of LKH time to PHGA time increased as the instance size increased, reaching 7.3 for the instance `pla33810`. For the instance `pla85900`, since the running time of LKH is roughly ten times that of LKH $-$ $.1N$, our method would be 18 times faster than LKH.

D. Asymptotic Comparison of PHGA and LKH

Johnson and McGeoch [20] generated a testbed for studying the asymptotic behavior of TSP algorithms and tested many TSP methods using this testbed. In this section, we compared our method with LKH using this testbed so that: 1) the asymptotic behaviors of PHGA and LKH can be compared and 2) our method can be compared with those studied in [20]. The testbed contains four classes of instances.

1) Random uniform Euclidean instances (uniform). Cities in these instances are points in a plane whose two coordinates are integers chosen uniformly from the interval $[0, 10^6)$. There are 26 such instances, with sizes increasing by a factor of $10^{0.5}$ from

TABLE IV
COMPARATIVE RESULTS OF PHGA AND LKH FOR TSPLIB INSTANCES WITH $N \geq 1000$

Name (Optimal/LB)	Method	Best (%)	Average (%)	Worst (%)	St. Dev.	Gen.	*N*opt	Time (s)	*P* (t-test)
dsj1000	PHGA	Optimal	18,659,712.2 (0.000)	18,659,930 (0.001)	76.5	3,060	9/10	2,860.7	–
(18,659,688)	LKH	Optimal	18,659,712.2 (0.000)	18,659,930 (0.001)	76.5	1,000	9/10	88.2	
pr1002	PHGA	Optimal	Optimal	Optimal	0.0	2,190	10/10	507.3	–
(259,045)	LKH	Optimal	Optimal	Optimal	0.0	1,002	10/10	25.9	
si1032	PHGA	Optimal	Optimal	Optimal	0.0	2,240	10/10	321.2	0.151
(92,650)	LKH	Optimal	92,663.2 (0.014)	92,716 (0.071)	27.8	1,032	8/10	35.4	
u1060	PHGA	Optimal	Optimal	Optimal	0.0	2,490	10/10	1,037.5	0.002
(224,094)	LKH	Optimal	224,110.2 (0.007)	224,121 (0.012)	13.9	1,060	4/10	119.6	
vm1084	PHGA	Optimal	Optimal	Optimal	0.0	3,220	10/10	1,089.4	0.151
(239,297)	LKH	Optimal	239,319.0 (0.009)	239,407 (0.046)	46.4	1,084	8/10	48.3	
pcb1173	PHGA	Optimal	Optimal	Optimal	0.0	2,530	10/10	419.2	0.330
(56,892)	LKH	Optimal	56,892.5 (0.001)	56,897 (0.009)	1.6	1,173	9/10	42.1	
d1291	PHGA	Optimal	Optimal	Optimal	0.0	2,520	10/10	813.1	0.025
(50,801)	LKH	Optimal	50,835.0 (0.067)	50,886 (0.167)	43.9	1,291	4/10	134.1	
rl1304	PHGA	Optimal	Optimal	Optimal	0.0	2,120	10/10	651.5	0.108
(252,948)	LKH	Optimal	253,030.0 (0.032)	253,335 (0.153)	153.1	1,304	7/10	49.3	
rl1323	PHGA	Optimal	Optimal	Optimal	0.0	3,080	10/10	1,193.1	1.1×10^{-5}
(270,199)	LKH	Optimal	270,234.5 (0.013)	270,254 (0.020)	18.7	1,323	1/10	48.1	
nrw1379	PHGA	Optimal	Optimal	Optimal	0.0	3,230	10/10	870.0	2.9×10^{-5}
(56,638)	LKH	Optimal	56,641.7 (0.007)	56,643 (0.009)	2.1	1,379	1/10	68.9	
fl1400	PHGA	Optimal	Optimal	Optimal	0.0	2,110	10/10	7,544.3	0.000
(20,127)	LKH	20,164 (0.184)	20,165.5 (0.191)	20,167 (0.199)	1.6	1,400	0/10	354.6	
u1432	PHGA	Optimal	Optimal	Optimal	0.0	2,400	10/10	739.3	–
(152,970)	LKH	Optimal	Optimal	Optimal	0.0	1,432	10/10	114.4	
fl1577	PHGA	Optimal	22,251.0 (0.009)	22,254 (0.022)	2.6	3,880	6/10	8,435.6	1.2×10^{-6}
(22,249)	LKH	22,251 (0.009)	22,261.0 (0.054)	22,263 (0.063)	3.6	1,577	0/10	659.8	
d1655	PHGA	Optimal	Optimal	Optimal	0.0	2,990	10/10	1,258.9	–
(62,128)	LKH	Optimal	Optimal	Optimal	0.0	1,655	10/10	105.3	
vm1748	PHGA	Optimal	Optimal	Optimal	0.0	2,890	10/10	1,587.7	0.030
(336,556)	LKH	Optimal	336,607.5 (0.015)	336,701 (0.043)	69.1	1,748	6/10	127.2	
u1817	PHGA	Optimal	Optimal	Optimal	0.0	3,420	10/10	1,638.6	6.3×10^{-8}
(57,201)	LKH	57,225 (0.042)	57,255.9 (0.096)	57,276 (0.131)	19.8	1,817	0/10	173.5	
rl1889	PHGA	Optimal	Optimal	Optimal	0.0	2,830	10/10	1,483.5	0.058
(316,536)	LKH	Optimal	316,539.4 (0.001)	316,549 (0.004)	5.3	1,889	6/10	119.2	
d2103	PHGA	Optimal	80,450.5 (0.001)	80,455 (0.006)	1.6	4,560	9/10	6,841.3	5.9×10^{-6}
(80,450)	LKH	80,457 (0.009)	80,473.5 (0.029)	80,486 (0.045)	11.4	2,103	0/10	492.6	
u2152	PHGA	Optimal	Optimal	Optimal	0.0	3,490	10/10	1,411.6	0.004
(64,253)	LKH	Optimal	64,279.2 (0.041)	64,314 (0.095)	25.3	2,152	2/10	304.9	
u2319	PHGA	234,338 (0.035)	234,338 (0.035)	234,338 (0.035)	0.0	2,820	0/10	3,200.5	
(234,256)	LKH	Optimal	Optimal	Optimal	0.0	2,319	10/10	957.5	0.000
pr2392	PHGA	Optimal	Optimal	Optimal	0.0	3,060	10/10	947.8	–
(378,032)	LKH	Optimal	Optimal	Optimal	0.0	2,392	10/10	275.0	
pcb3038	PHGA	Optimal	Optimal	Optimal	0.0	4,120	10/10	2,243.7	0.048
(137,694)	LKH	Optimal	137,712.7 (0.014)	137,753 (0.043)	27.9	3,038	5/10	683.6	
fl3795	PHGA	Optimal	Optimal	Optimal	0.0	3,470	10/10	7,859.4	0.053
(28,772)	LKH	Optimal	28,792.7 (0.072)	28,881 (0.379)	27.9	3,795	1/10	3,475.3	
fnl4461	PHGA	Optimal	182,568.7 (0.001)	182,575 (0.005)	3.2	6,510	5/10	2,568.6	0.627
(182,566)	LKH	Optimal	182,569.4 (0.002)	182,575 (0.005)	3.2	4,461	4/10	1,405.6	
rl5915	PHGA	565,543 (0.002)	565,550.8 (0.004)	565,559 (0.005)	7.4	5,390	0/10	4,413.3	4.5×10^{-5}
(565,530)	LKH	565,561 (0.005)	565,667.7 (0.024)	565,768 (0.042)	68.9	5,915	0/10	2,420.1	
rl5934	PHGA	Optimal	556,049.1 (0.001)	556,081 (0.006)	11.3	5,560	8/10	4,073.0	2.0×10^{-7}
(556,045)	LKH	556,164 (0.021)	556,508.9 (0.083)	556,872 (0.149)	178.9	5,934	0/10	3,089.8	
pla7397	PHGA	Optimal	23,260,805.4 (0.000)	23,260,814 (0.000)	27.2	5,700	1/10	8,966.6	0.162
(23,260,728)	LKH	Optimal	23,261,418.0 (0.003)	23,263,996 (0.014)	1329.3	7,397	5/10	16,107.5	
rl11849	PHGA	923,318 (0.003)	923,355.7 (0.007)	923,442 (0.017)	39.8	10,110	0/10	14,699.6	0.008
(923,288)	LKH	923,315 (0.003)	923,475.8 (0.020)	923,640 (0.038)	122.3	11,849	0/10	20,221.4	
usa13509	PHGA	19,982,874 (0.000)	19,984,260.5 (0.007)	19,985,149 (0.011)	725.8	13,950	0/10	20,537.7	0.157
(19,982,859)	LKH	19,983,880 (0.005)	19,984,750.6 (0.009)	19,986,465 (0.018)	757.0	13,509	0/10	29,255.5	
brd14051	PHGA	469,431 (0.012)	469,451.0 (0.016)	469,482 (0.023)	16.0	18,290	0/10	24,685.8	
(469,374)	LKH	469,409 (0.007)	469,429.7 (0.012)	469,458 (0.018)	13.7	14,051	0/10	46,570.6	0.005
d15112	PHGA	1,573,183 (0.006)	1,573,330.4 (0.016)	1,573,406 (0.020)	81.5	21,170	0/10	34,211.4	
(1,573,084)	LKH	1,573,168 (0.005)	1,573,273.1 (0.012)	1,573,357 (0.017)	59.5	15,112	0/10	56,278.5	0.089
d18512	PHGA	645,268 (0.011)	645,323.8 (0.019)	645,366 (0.026)	36.3	21,990	0/10	35,985.6	0.551
(645,198)	LKH	645,305 (0.017)	645,332.2 (0.021)	645,372 (0.027)	24.3	18,512	0/10	109,127.0	
pla33810	PHGA	66,062,314 (0.087)	66,067,083.0 (0.094)	66,072,371 (0.102)	3,472.5	24,770	0/10	159,580.3	8.4×10^{-6}
(66,005,185)	LKH	66,076,272 (0.108)	66,086,687.7 (0.123)	66,102,893 (0.148)	9,473.1	33,810	0/10	1,163,388.0	
pla85900	PHGA	142,408,271 (0.071)	142,415,548.6 (0.076)	142,422,834 (0.081)	4,427.6	67,420	0/10	397,507.5	–
(142,307,500)	LKH-.1N	142,600,908 (0.206)	142,600,908.0 (0.206)	142,600,908 (0.206)	0.0	8,590	0/1	721,693.0	

PHGA times were normalized to a 933-MHz Pentium III PC based on the speedups in Table III. LKH times were measured on a 933-MHz Pentium III PC.

TABLE V
COMPARISON OF PHGA AND LKH ON TOUR QUALITY

Method	Class	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$
PHGA	U	**0.736**	**0.705**	**0.676**	**0.673**	0.684	0.741
	C	**0.538**	**0.614**	**0.685**	**0.706**	0.751	0.906
	T	**0.956**	**0.858**	0.540	**0.555**	0.430	
	M	0.208	0.673				
LKH	U	0.740	0.708	0.680	0.681	–	–
	C	0.556	0.615	0.782	1.460	–	–
	T	0.958	0.860	**0.536**	0.571	–	
	M	**0.016**	**0.004**				

TABLE VI
COMPARISON OF PHGA AND LKH ON RUNNING TIME(S)

Method	Class	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$
PHGA	U	673	2,753	20,293	**87,408**	563,481	5,016,759
	C	5,822	10,474	34,071	**109,411**	866,895	5,616,118
	T	635	2,208	**21,842**	**150,877**	343,568	
	M	6,395	24,849				
LKH	U	**33**	**636**	**14,295**	517,374	–	–
	C	**75**	**1,143**	**12,774**	186,522	–	–
	T	**45**	**784**	33,168	1,164,534	–	
	M	**24**	**397**				

PHGA times were normalized to a 933-MHz Pentium III PC based on the speedups in Table III. For each problem size except $N = 10^{5.5}$, the speedup was the average of speedups of all instances representing this size. For $N = 10^{5.5}$, the speedup of $N = 10^5$ was used. The speedups are 7.65, 7.46, 7.48, 7.10, 8.03, and 8.03, respectively. LKH times were measured on a 933-MHz Pentium III PC.

$N = 10^3$ to $N = 10^7$. These instances are designed for studying the asymptotic behavior of TSP algorithms.

2) Random clustered Euclidean instances (clustered). In these instances, cities are clustered by $N/100$ centers whose coordinates are chosen uniformly from the interval $[0, 10^6)$. There are 23 instances with sizes increasing by a factor of $10^{0.5}$ from $N = 10^3$ to $N = 10^{5.5}$. These instances are designed to be challenging TSP local searches.
3) TSPLIB instances. This class includes all 34 instances having 1000 or more cities in TSPLIB. Johnson and McGeoch used the four instances `pr1002`, `pcb1173`, `rl1304`, and `nrw1379` for the size $N = 10^3$; the three instances `pr2392`, `pcb3038`, and `fnl4461` for $N = 10^{3.5}$; the two instances `pla7397` and `brd14051` for $N = 10^4$; the instance `pla33810` for $N = 10^{4.5}$; and the instance pla85900 for $N = 10^5$. These instances can be viewed as "real world" problems.
4) Random matrix instances (matrix). Distances in these instances are integers chosen uniformly in the interval $[0, 10^6)$. There are seven instances ranging in size from $N = 10^3$ to $N = 10^4$. Although these instances have no direct application to practice, they offer a great challenge to many heuristics and thus are useful in studying the robustness of TSP algorithms.

We excluded all uniform instances having more than $10^{5.5}$ cities and the 10^4-city matrix instance, as our machines did not have enough memory to handle them. Moreover, we did not apply LKH to the instances with 10^5 or more cities since it would have taken too much time. Following Johnson and McGeoch [20], only one run was performed for each remaining instance. Therefore, the results are not statistically significant, and they should be considered only as a qualitative comparison between the two algorithms.

Summaries of the comparative results of PHGA and LKH on tour quality and running time are given in Tables V and VI, respectively. Note that the tour qualities in Table V are given as percentages over the Held–Karp (HK) lower bounds [13], [14]. These HK bounds were computed by the CONCORDE code [20]. The notations U, C, T, and M in column "Class" denote uniform, clustered, TSPLIB, and matrix classes, respectively.

Most of the LKH entries in Table V match quite well with the results reported in [20]. However, the average percentage excess over the HK bound of LKH for the two $10^{4.5}$-city clustered instances (`C31k.0` and `C31k.1`) was reported as 0.53%, while in our results it was 1.46%. In our experiments, LKH produced a very poor quality solution for the instance `C31k.1` (2.24% as opposed to 0.68% for `C31k.0`). Therefore, we reran LKH for this instance with a different seed, and the result was better (1.24%). However, this result was still far from the one reported in [20]. Further investigation revealed that the LKH result for `C31k.0` was mistakenly reported as another result of `C31k.1`, and the correct tour length for `C31k.0` was 59 553 017 [16]. Therefore, the tour quality for this entry of LKH in [20] should be 0.75%.

From the viewpoint of tour quality (Table V), PHGA wins over LKH for all sizes in the uniform and clustered classes. PHGA also wins for all but the size $N = 10^4$ in the TSPLIB class. Although the differences shown in the table seem small, they likely account for a large portion of the gap between the LKH tours and the optimal tours. Indeed, for uniform instances, PHGA found optimal tours for all instances of size $N = 10^3$ and three out of five instances of size $N = 10^{3.5}$. It also found optimal tours for all clustered and TSPLIB instances of sizes $N = 10^3$ and $N = 10^{3.5}$. PHGA, however, performs much worse than LKH on matrix instances. Note that most of the heuristics investigated in [20] failed to produce solutions within 1% above of the HK lower bounds for these matrix instances. Moreover, if these instances were ever to arise in practice, exact algorithms such as CONCORDE should be the first option. The CONCORDE code was able to solve to optimality all matrix instances in their testbed using its default parameters with the running time only marginally more than LKH [20].

From the viewpoint of running time (Table VI), PHGA was slower than LKH for all TSP classes of size $N = 10^3$ and $N = 10^{3.5}$. However, PHGA was comparable with LKH when $N = 10^4$, with PHGA slightly faster for TSPLIB instances and slightly slower for uniform and clustered instances. For $N = 10^{4.5}$, PHGA was clearly faster: it was approximately six, two, and seven times faster than LKH for uniform, clustered, and TSPLIB instances, respectively. For uniform instances, the asymptotic running times of PHGA and LKH as measured by our machines were $O(N^{1.5})$ and $O(N^{2.8})$, respectively.

E. Comparison of PHGA With Other Methods

A comparison of PHGA with other hybrid GAs is given in Table VII. The results for other methods except LKH are taken from the literature. PHGA was the most effective method in terms of tour quality. Although it is difficult to compare the CPU times of these methods, as different machine types were used, PHGA was not too slow (and possibly faster) compared to other methods that have approximately the same level of tour quality (e.g., GLS-100, GA-EAX, and Cga-LK).

As mentioned in Section III-B, the use of a more powerful local search is one of the main reasons for the improved performance of our method. The results in Table VII confirm this fact. Both PHGA and LKH employ LKs with 5-Opt basic moves, and their results were also much better than hybrid GAs that use standard LKs. However, our LK search engine is not as powerful as the one in LKH (since it does not search for nonsequential moves), but the tours obtained by PHGA are often better than the tours obtained by LKH. This indicates that our method may have a better choice of GA model and/or balance between the global and local searches.

TABLE VII
COMPARATIVE RESULTS OF PHGA AND OTHER HYBRID GAS

Name (Optimal/LB)	Method [a]	Average (%)	N_{opt}	Time (s) [b]
pcb3038	PHGA	**Optimal**	10/10	2,244
(137,694)	LKH	137,712.7 (0.014)	5/10	684
	GLS-100	137,702.6 (0.006)	3/30	6,955
	GA-EAX	– (0.001)	34/50	1,129
	NGA	137,765.0 (0.057)	0/100	816
fl3795	PHGA	**Optimal**	10/10	7,859
(28,772)	LKH	28,792.7 (0.072)	1/10	3,475
	GLS-100	28,794.7 (0.079)	1/30	7,212
	Asparagos	28,820.0 (0.167)	0/10	61,080
	Cga-LK	**Optimal**	5/5	5,033
fnl4461	PHGA	**182,568.7 (0.001)**	5/10	2,569
(182,566)	LKH	182,569.4 (0.002)	4/10	1,406
	GLS-40	183,047.1 (0.263)	0/10	742
	GA-EAX	– (**0.001**)	29/50	4,730
	Cga-LK	182,578.4 (0.007)	≥1/5	33,887
rl5915	PHGA	**565,550.8 (0.004)**	0/10	4,413
(565,530)	LKH	565,667.7 (0.024)	0/10	2,420
	Cga-LK	565,554.0 (0.004)	≥1/5	30,935
usa13509	PHGA	**19,984,260.5 (0.007)**	0/10	20,538
(19,982,859)	LKH	19,984,705.6 (0.009)	0/10	29,256
	GLS-40	20,057,767.0 (0.375)	0/10	6,638
	Cga-LK	19,991,585.0 (0.043)	0/1	493,200
pla33810	PHGA	**66,067,083.0 (0.094)**	0/10	159,580
(66,005,185)	LKH	66,086,687.7 (0.123)	0/10	1,163,388
	GLS-20	66,321,344.7 (0.479)	0/10	11,523
pla85900	PHGA	**142,415,548.6 (0.076)**	0/10	397,508
(142,307,500)	LKH-.1N	142,600,908.0 (0.206)	0/1	721,693
	GLS-20	142,986,675.5 (0.477)	0/10	52,180

[a] The suffix numbers after the GLS method show the opulation size. For GA-EAX, results with a population size of 300 were used.

[b] CPU times of PHGA were normalized to a 933-MHz Pentium III PC based on the speedups in Table III. LKH times were measured on a 933-MHz Pentium III PC; GLS on a 500-Mhz Pentium III PC; Asparagos on a 170-MHz SUN UltraSparc; GA-EAX on a 1700-MHz Xeon PC; NGA on a 450-MHz Pentium III PC; and Cga-LK on a 350-Mhz Pentium III PC.

Johnson and McGeoch [20] proposed a way to roughly compare the CPU times of various TSP algorithms on different machine configurations. They provided a benchmark code for the greedy heuristic (available at the TSP Challenge homepage). Participants should report the CPU times of this benchmark code on their machines for a set of instances covering all instance sizes that can be handled. In order for our method to be compared with those reported in [20] and in future works, we measured the CPU times of the benchmark greedy code on one of our cluster PCs. The CPU times were 11 s for $N = 10^3$, 11 s for $N = 10^{3.5}$, 20 s for $N = 10^4$, 35 s for $N = 10^{4.5}$, 43 s for $N = 10^5$, and 48 s for $N = 10^{5.5}$.

F. Solving the World TSP Challenge

To further demonstrate the effectiveness of our method, we applied it to the 1 904 711-city World TSP Challenge (available at http://www.tsp.gatech.edu/world). Since our machines did not have enough memory for PHGA to handle the instance directly, we used the following strategy.

Step 1) Perform 20 runs of ILK $-$ .1N. Choose the best tour of these 20 runs and apply ILK $-$ 3N to it.

Step 2) Based on the current best tour, divide the world instance into a number of smaller subinstances. This is done by partitioning the current best tour into a number (from six to 300) of segments of roughly equal size. Clearly, we can improve the tour if, for at least one of these segments, we can find a shorter path starting from one end point of the segment, going through all nodes in the segments, and ending at the other end point. Thus, for each segment, a subinstance can be formed by finding the shortest tour going through all nodes of this segment, with the only constraint that the tour must contain the edge connecting the two end points of the segment. This step is done automatically by our program with the inputs being the current best tour and the number of subinstances.

Step 3) Apply PHGA to these subinstances. For each subinstance, if the new segment (taken by removing the edge between the two end points from the new tour produced by PHGA) is shorter than the previous segment, replace it.

Step 4) Reconnect all the best segments of each subinstance to form a new best tour for the World instance. This step is also done automatically by our program.

Step 5) Apply ILK $-$ N to the new best tour.

Step 6) Repeat Step 2) until the computation time runs out or the answer is satisfactory.

Steps 1) and 5) were performed on a Pentium IV 1.6-GHz PC. Step 3) was performed using a cluster of 32 machines (Pentium III, 933 MHz), an ULTRA 80 with four processors, an ULTRA 60 with two processors, and the Pentium IV PC mentioned above. After ∼50 days, a tour of length 7 518 425 642 m had been obtained. However, all of the machines did not run continuously during this period, and unfortunately, we did not record the CPU time for each individual run, so we cannot specify the exact CPU time usage. Our tour (completed on June 2, 2003) improves the previous best known tour of length 7 518 528 361 m found by Helsgaun using a variant of his LKH. The exact CPU time usage for obtaining Helsgaun's tour is not available [16]. Our tour can still be improved further by using the above procedure: in the last days of our search process, we were able to reduce the tour length by an average of 15 000 m/day.

IV. CONCLUSION

We have described a hybrid GA aimed at solving large-scale TSPs. The method is based on a multipopulation steady-state GA and combined with the local search heuristics. It uses a variant of the MPX crossover and the double-bridge mutation. An LK implementation that employed some recently proposed improvements for LK was used to improve the offspring produced by the GA's genetic operators. The method can be run in parallel on a multiprocessor machine or on a cluster of PCs to obtain solutions more quickly.

Our method was validated by comparing it with LKH, which is one of the most effective heuristics for the TSP. Experimental results of benchmarks having up to 316 228 cities show that our method works more effectively and efficiently when solving large-scale TSPs with 10 000 or more cities. We also applied our method, together with an implementation of the iterated LK, to find a new best tour for the 1 904 711-city World TSP Challenge.

The main contribution of our correspondence is to show that, when properly implemented, the combination of a GA with local search is very promising for the TSP, and that the effectiveness and efficiency of the local search play important roles in the performance of hybrid GAs. The design of the GA and the balance between local and global search also contribute to the improvement of hybrid GAs.

ACKNOWLEDGMENT

The authors would like to thank D. Whitley, K. Helsgaun, D. Applegate, W. Cook, A. Rohe, G. Reinelt, D. Johnson, L. McGeoch, and others for providing program codes and benchmarks.

REFERENCES

[1] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, "TSP cuts which do not conform to the template paradigm," in *Computational Combinatorial Optimization*. New York: Springer-Verlag, 2001, pp. 261–304.

[2] D. Applegate, W. Cook, and A. Rohe, "Chained Lin–Kernighan for large traveling salesman problems," *INFORMS J. Comput.*, vol. 15, no. 1, pp. 82–92, Jan. 2003.

[3] R. Baraglia, J. I. Hidalgo, and R. Perego, "A hybrid heuristic for the traveling salesman problem," *IEEE Trans. Evol. Comput.*, vol. 5, no. 6, pp. 613–622, Dec. 2001.

[4] J. L. Bentley, "Experiments on traveling salesman heuristics," in *Proc. 1st Annu. ACM-SIAM Symp. Discr. Algorithms,* D. Johnson, Ed., 1990, pp. 91–99.

[5] E. Cantu-Paz, "Implementing fast and flexible parallel genetic algorithms," in *Practical Handbook of Genetic Algorithms*, vol. III, L. D. Chambers, Ed. Boca Raton, FL: CRC, 1998, pp. 65–84.

[6] J. Dzubera and D. Whitley, "Advanced correlation analysis of operators for the traveling salesman problem," in *Parallel Problem Solving From Nature—PPSN III*, Y. Davidor, H.-P. Schwefel, and R. Männer, Eds. New York: Springer-Verlag, 1994, pp. 68–77.

[7] M. L. Fredman, D. S. Johnson, L. A. McGeoch, and G. Ostheimer, "Data structures for traveling salesmen," *J. Alg.*, vol. 18, no. 3, pp. 432–479, May 1995.

[8] B. Freisleben and P. Merz, "A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems," in *Proc. IEEE Int. Conf. Evol. Comput.*, 1996, pp. 616–621.

[9] D. E. Goldberg and R. Lingle, "Alleles, loci, and the traveling salesman problem," in *Proc. Int. Conf. Genetic Algorithms and Their Appl.*, 1985, pp. 154–159.

[10] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[11] M. Gorges-Schleuter, "Asparagos96 and the traveling salesman problem," in *Proc. IEEE Int. Conf. Evol. Comput.*, 1997, pp. 171–174.

[12] J. J. Grefenstette, R. Gopal, B. Rosmaita, and D. VanGucht, "Genetic algorithms for the traveling salesman problem," in *Proc. Int. Conf. Genetic Algorithms and Their Appl.*, 1985, pp. 160–168.

[13] M. Held and R. M. Karp, "The traveling salesman problem and minimal spanning trees," *Oper. Res.*, vol. 18, no. 6, pp. 1138–1162, 1970.

[14] ——, "The traveling salesman problem and minimal spanning trees: Part II," *Math. Program.*, vol. 1, no. 1, pp. 6–25, 1971.

[15] K. Helsgaun, "An effective implementation of the Lin–Kernighan traveling salesman heuristic," *Eur. J. Oper. Res.*, vol. 126, no. 1, pp. 106–130, Oct. 2000.

[16] ——, Department of Computer Science, Roskilde Univ., Roskilde, Denmark, private communication, 2004.

[17] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor: Univ. of Michigan Press, 1975.

[18] H. Ishibuchi, T. Yoshida, and T. Murata, "Balance between genetic search and local search in memetic algorithms for multiobjective permutation flowshop scheduling," *IEEE Trans. Evol. Comput.*, vol. 7, no. 2, pp. 204–223, Apr. 2003.

[19] S. Johnson and L. A. McGeoch, "The traveling salesman problem: A case study in local optimization," in *Local Search in Combinatorial Optimization*. Hoboken, NJ: Wiley, 1997, pp. 215–310.

[20] ——, "Experimental analysis of heuristics for the STSP," in *The Traveling Salesman Problem and Its Variations*. Norwell, MA: Kluwer, 2002, pp. 369–443.

[21] S. Jung and B. Moon, "The natural crossover for the 2D Euclidean TSP," in *Proc. Genetic and Evol. Comput. Conf.*, 2001, pp. 1003–1010.

[22] N. Krasnogor and J. E. Smith, "A tutorial for competent memetic algorithms: Model, taxonomy, and design issues," *IEEE Trans. Evol. Comput.*, vol. 9, no. 5, pp. 474–488, Oct. 2005.

[23] S. Lin and B. Kernighan, "Effective heuristic algorithm for the traveling salesman problem," *Oper. Res.*, vol. 21, no. 2, pp. 498–516, 1973.

[24] O. Martin, S. W. Otto, and E. W. Felten, "Large-step Markov chains for the traveling salesman problem," *Complex Syst.*, vol. 5, no. 3, pp. 299–326, 1991.

[25] K. Mathias and D. Whitley, "Genetic operators, the fitness landscape, and the traveling salesman problem," in *Parallel Problem Solving From Nature*. Amsterdam, The Netherlands: Elsevier, 1992, pp. 219–228.

[26] P. Merz and B. Freisleben, "Genetic local search for the TSP: New results," in *Proc. IEEE Int. Conf. Evol. Comput.*, 1997, pp. 159–164.

[27] ——, "Memetic algorithms for the traveling salesman problem," *Complex Syst.*, vol. 13, no. 4, pp. 297–345, 2001.

[28] H. Muhlenbein, "Evolution in time and space—The parallel genetic algorithm," in *Foundations of Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann, 1991, pp. 316–337.

[29] Y. Nagata and S. Kobayashi, "Edge assembly crossover: A high-power genetic algorithm for the traveling salesman problem," in *Proc. 7th Int. Conf. Genetic Algorithms*, 1997, pp. 450–457.

[30] Y. Nagata, "The EAX algorithm considering diversity loss," in *Parallel Problem Solving From Nature—PPSN VIII*. New York: Springer-Verlag, 2004, pp. 332–341.

[31] H. D. Nguyen, I. Yoshihara, and M. Yasunaga, "Modified edge recombination operators of genetic algorithms for the traveling salesman problem," in *Proc. 3rd Asia-Pacific Conf. Simul. Evol. and Learn.*, Nagoya, Japan, 2000, pp. 2815–2820.

[32] H. D. Nguyen, I. Yoshihara, K. Yamamori, and M. Yasunaga, "Greedy genetic algorithms for symmetric and asymmetric TSPs," *IPSJ Trans. Math. Modeling and Appl.*, vol. 43, no. SIG10 (TOM7), pp. 165–175, Nov. 2002.

[33] H. D. Nguyen, "Hybrid genetic algorithms for combinatorial optimization problems," Ph.D. dissertation, Faculty Eng., Univ. Miyazaki, Miyazaki, Japan, 2004.

[34] I. Oliver, D. Smith, and J. Holland, "A study of permutation crossover operators on the traveling salesman problem," in *Proc. 2nd Int. Conf. Genetic Algorithms*, 1987, pp. 224–230.

[35] Y. S. Ong and A. J. Keane, "Meta-Lamarckian learning in memetic algorithms," *IEEE Trans. Evol. Comput.*, vol. 8, no. 2, pp. 99–110, Apr. 2004.

[36] G. Reinelt, "TSPLIB—A traveling salesman problem library," *ORSA J. Comput.*, vol. 3, no. 4, pp. 376–384, 1991. [Online]. Available: http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95

[37] H. Sengoku and I. Yoshihara, "A fast TSP solver using GA on JAVA," in *Proc. 3rd Int. Symp. Artif. Life and Robot.*, 1998, pp. 283–288.

[38] T. Starkweather, S. McDaniel, K. Mathias, C. Whitley, and D. Whitley, "A comparison of genetic sequencing operators," in *Proc. 4th Int. Conf. Genetic Algorithms*, 1991, pp. 69–76.

[39] A. Takeda, S. Yamada, K. Sugawara, I. Yoshihara, and K. Abe, "Optimization of delivery route in a city area using genetic algorithm," in *Proc. 4th Int. Symp. Artif. Life and Robot.*, 1999, pp. 496–499.

[40] D. Whitley, "The GENITOR algorithm and selective pressure: Why rank-based allocation of reproductive trials is best," in *Proc. 3rd Int. Conf. Genetic Algorithms*, 1989, pp. 116–121.

[41] D. Whitley, T. Starkweather, and D. Fuquay, "Scheduling problems and traveling salesman: The genetic edge recombination operator," in *Proc. 3rd Int. Conf. Genetic Algorithms*, 1989, pp. 133–140.

[42] D. Whitley, T. Starkweather, and D. Shaner, "The traveling salesman and sequence scheduling: Quality solutions using genetic edge recombination," in *The Handbook of Genetic Algorithms*, L. Davis, Ed. New York: Van Nostrand, 1991, pp. 350–372.