Brief Papers

A Hybrid Heuristic for the Traveling Salesman Problem

R. Baraglia, J. I. Hidalgo, and R. Perego

***Abstract*—The combination of genetic and local search heuristics has been shown to be an effective approach to solving the traveling salesman problem (TSP). This paper describes a new hybrid algorithm that exploits a compact genetic algorithm in order to generate high-quality tours, which are then refined by means of the Lin–Kernighan (LK) local search. Local optima found by the LK local search are in turn exploited by the evolutionary part of the algorithm in order to improve the quality of its simulated population. The results of several experiments conducted on different TSP instances with up to 13 509 cities show the efficacy of the symbiosis between the two heuristics.**

***Index Terms*—Compact genetic algorithms, hybrid GA, Lin–Kernighan algorithm, TSP.**

I. Introduction

THE TRAVELING salesman problem (TSP) is probably the most famous combinatorial optimization problem. Given a finite set of cities $C = \{c_1, c_2, \ldots, c_k\}$ and a distance $d(c_i, c_j)$, $i \neq j$, between each pair, the TSP asks for finding a tour through all of the cities, visiting each exactly once, and returning to the originating city such that the total distance traveled is minimized.

Because of its simplicity and applicability to many fields, the TSP has often served as a testing ground for many exact and heuristic optimization algorithms [1]–[3]. Highly optimized exact algorithms based on the branch-and-cut method [2], [4] have been proposed that enable even large TSP instances to be solved. The computational demand of exact approaches however is huge. Some heuristic approaches, on the other hand, have been proved to be very effective both in terms of execution times and quality of the solutions achieved. A broad taxonomy of TSP heuristics distinguishes between local search approaches exploiting problem-domain knowledge and classical problem-independent heuristics.

Domain-specific heuristics, such as 2-Opt [5], 3-Opt [6], and Lin–Kernighan (LK) [7], are surprisingly very effective for the TSP. 2-Opt repeatedly applies a simple move that breaks the current solution in two parts and reconnects the tour in the other possible way.[1] If the move reduces the length of the tour, the modified tour becomes the current solution. The algorithm stops when a locally optimal tour is found for which no other move yields an improvement. 3-Opt works similarly, but it breaks the tour in three parts instead of two.

LK essentially uses 2-Opt and 3-Opt moves from within a tabu search algorithmic framework [8], [9]. It is considered as the heuristic that leads to the best solutions and, thus, the benchmark against which all other heuristics should be tested. Moreover, very efficient implementations have been devised for LK that take just a few seconds to compute a high-quality solution for problems with hundreds of cities [3], [10]. Computational efficiency is very important since LK must be executed several times within an overall tour-finding schema in order to explore different regions of the search space, thus, improving the quality of the final solution.

On the other hand, general problem-independent heuristics like simulated annealing (SA) [11] and genetic algorithms (GAs) [12]–[14] perform quite poorly on large TSP instances. They require high execution times for solutions whose quality is often not comparable with those achieved in much less time by their domain-specific local search counterparts.

Crossover operators that preserve in the offsprings parts of the parents' tours and inversion operators that resemble a k-Opt move have been proposed to improve the effectiveness of GAs for the TSP [14]. Unfortunately, edge-preserving crossover operators are computationally expensive, while inversion-based algorithms difficultly escape local optima. A good tradeoff between the two approaches is the Tao and Michalewicz's [15] *inver-over* operator that tries to combine the computational efficiency of inversion with the power of crossover. The inver-over operator applies to the selected tour a sequence of simple inversions (i.e., 2-Opt moves), where each inversion may insert in the new individual an edge of another randomly selected individual. Tao and Michalewicz's algorithm deals very efficiently with small TSP instances. Nevertheless, it cannot still compete with domain-specific approaches on large TSPs.

Several published results demonstrate that combining a problem-independent heuristic with a local search method is a viable and effective approach for finding high-quality solutions of large TSPs. The problem-independent part of the hybrid algorithm drives the exploration of the search space, thus,

Manuscript received April 19, 2000; revised September 20, 2000 and January 15, 2000. The work of J. I. Hidalgo was supported by the Spanish Government under Grant TYC 1999-0474.

R. Baraglia and R. Perego are with the Istituto CNUCE, Consiglio Nazionale delle Ricerche (CNR), 56010 Pisa, Italy.

J. I. Hidalgo is with the Departamento de Arquitectura de Computadores y Automática, Universidad Complutense, 28040 Madrid, Spain.

Publisher Item Identifier S 1089-778X(01)03465-8.

[1]More specifically, given a tour $(c_1, \ldots, c_{p-1}, c_p, \ldots, c_q, c_{q+1}, \ldots, c_k)$, a 2-Opt move applied to edges (c_{p-1}, c_p) and (c_q, c_{q+1}) inverts the subsequence $(c_p, \ldots, c_q)$ producing the new tour $(c_1, \ldots, c_{p-1}, c_q, c_{q-1}, \ldots, c_{p+1}, c_p, c_{q+1}, \ldots, c_k)$.

focusing on the global optimization task, while the local search algorithm visits the promising subregions of the solution space.

Martin and Otto [16], [17] proposed the *chained local optimization* algorithm, where a special type of 4-Opt move is used under the control of a SA schema to escape from the local optima found with LK.

Freisleben and Merz [18] designed genetic operators to search the space of the local optima determined with LK. They used a specific crossover operator that preserves the edges found in both the parents.

Georges-Schleuter [19] instead tested the exploitation of simple k-Opt moves within her *Asparagos96* parallel GA. She concluded that, for large problem instances, the strategy of producing many fast solutions might be almost as effective as using powerful local search methods with fewer solutions. Unfortunately, the last two proposals fail to clearly demonstrate the contribution of genetics to the results achieved.

In this paper, we propose *Cga-LK* as a new hybrid algorithm for the resolution of the TSP. Cga-LK combines an original compact genetic algorithm (Cga) with an efficient implementation of LK. The term *compact* derives from the fact that our algorithm does not manage a population of solutions but only mimics its existence. The idea behind this algorithm arose from the study of the Cga proposed by Harik *et al.* [20]. They showed that the Cga mimics the order-one behavior of a simple GA (sGA) with a given population size and selection rate. We extended such study and obtained promising results by applying the Cga to a "difficult" order-k optimization problem such as the TSP. Moreover, we exploited the low memory requirements and the high computational efficiency of the Cga to design Cga-LK.

In Cga-LK, the Cga is used to explore the more promising part of the TSP solution space in order to generate "good" initial solutions, which are refined with LK. The refined solutions are also exploited to improve the quality of the simulated population as the execution progresses. We concentrated attention on the symmetric TSP and tested our algorithm on TSPLIB [21] instances with up to 13 509 cities. The main goal of our paper is to demonstrate, through a detailed experimental analysis, the good symbiotic behavior between genetics and the LK local search algorithm.

The paper is organized as follows. Section II describes the Cga exploited by our proposal. Some results obtained with genetics only on small TSP instances are also presented and discussed. Our hybrid approach is detailed in Section III. Section III-A presents the results achieved with our implementation on TSP instances ranging from 198 to 13 509 cities. The results are also compared with those of other algorithms applied to the same problem instances. Finally, Section IV outlines some conclusions.

II. Copmact Genetic Algorithm

The Cga [20] does not manage a population of solutions like traditional GAs [22], but instead only mimics its existence. The idea on which the Cga is based was primarily inspired by the *random walk* model, proposed to estimate GA convergence on a class of problems where there is no interaction between the building blocks constituting the solution [23]. Other concepts that inspired the Cga proposal were *bit-based simulated crossover* [24] and *population-based incremental learning* [25]. The Cga represents the population by means of a vector of values $p_i \in [0, 1]\ \forall i = 1, \ldots, l$, where l is the number of alleles needed to represent the solutions. Each value p_i measures the proportion of individuals in the simulated population that have a zero (one) in the ith locus of their representation. By treating these values as probabilities, new individuals can be generated and, based on their fitness, the probability vector updated in order to favor the generation of better individuals.

The values for probabilities p_i are initially set to 0.5 to represent a randomly generated population in which the value for each allele has equal probability. At each iteration the Cga generates two individuals on the basis of the current probability vector and compares their fitness. Let W be the representation of the individual with a better fitness and L the individual whose fitness is worse. The two representations are used to update the probability vector at step $k+1$ in the following way:

$$p_i^{k+1} = \begin{cases} p_i^k + 1/n, & \text{if } w_i = 1 \wedge l_i = 0 \\ p_i^k - 1/n, & \text{if } w_i = 0 \wedge l_i = 1 \\ p_i^k, & \text{if } w_i = l_i \end{cases} \tag{1}$$

where n is the size of the population simulated and w_i (l_i) is the value of the ith allele of W (L). The Cga ends when the values of the probability vector are all equal to zero or one. At this point, vector p itself represents the final solution.

Note that the Cga evaluates an individual by considering its whole chromosome. At each iteration, some alleles of solution W might not belong to the optimal solution of the problem and the corresponding probability values may be modified wrongly.

When applied to order-one problems, a Cga is approximately equivalent to an sGA with uniform crossover: it achieves solutions of comparable quality with approximately the same number of fitness evaluations. To solve problems with higher order building blocks, GAs with both higher selection rates and larger population sizes have to be exploited [26]. The Cga selection pressure can be increased by modifying the algorithm in the following way: 1) at each iteration, generate s individuals from the probability vector instead of two; 2) choose among the s individuals the one with the best fitness and select as W its representation; and 3) compare W with each one of the other $s-1$ individuals and update the probability vector according to (1). The other parts of the algorithm remain unchanged. Such an increase to the selection pressure helps the Cga to converge to better solutions since it increases the survival probability of higher order building blocks [20].

Although the Cga mimics the order-one behavior of a GA with uniform crossover, it was not proposed as an alternative algorithm. According to its authors, it can be used to quickly assess the "difficulty" of a problem. A problem is easy if it can be solved with a Cga exploiting a low selection rate. The more the selection rate has to be increased to solve the problem, the more it should be considered as difficult. Moreover, given a population of n individuals, the Cga updates the probability vector by a constant value equal to $1/n$. Only $\log_2 n$ bits are, thus, needed to store the finite set of values for each p_i. The Cga, therefore, re-

```
Program TSP_Cga
  begin
    Initialize(P,method);
    F_best := INT_MAX;
    repeat
       S[1] := Generate(P);
       F[1] := Tour_Length(S[1]);
       idx_best := 1;
       for k := 2 to s do
          S[k] := Generate(P);
          F[k] := Tour_Length(S[k]);
          if (F[k] < F[idx_best]) then idx_best := k;
       end for
       for k := 1 to s do
          if (F[idx_best] < F[k]) then Update(P,S[idx_best],S[i]);
       end for
       if (F[idx_best] < F_best) then
          count := 0;
          F_best := F[idx_best];
          S_best := S[idx_best];
       else
          Update(P,S_best,S[idx_best]);
          count := count + 1;
       end if
    until (Convergence(P) OR count > CONV_LIMIT)
    Output(S_best,F_best);
  end
```

Fig. 1. Pseudocode of our Cga for the TSP.

quires $l \cdot \log_2 n$ bits compared to the $n \cdot l$ bits needed by a classic GA. Large population sizes can be, thus, exploited without significantly increasing memory requirements.

A. Cga for the TSP

In order to design a Cga for the TSP, we adopted the *path-representation* model, which represents a feasible tour as one of the $k!$ possible permutations of the k cities [14] and we considered the frequencies of the edges occurring in the simulated population. A $k \times k$ triangular matrix of probabilities P was used to this end. Each element $p_{i,j}$, $i \geq j$ of P represents the proportion of individuals whose tour contains edge (c_i, c_j). If n is the population size, our Cga, thus, requires $(k^2/2) \cdot \log_2 n$ bits to represent the population compared to the $k \cdot n \cdot \log_2 k$ bits required by a classical GA. Fig. 1 shows the pseudocode of our Cga. Its main functions are discussed in the following.

Initialization of the Probability Matrix: Two different methods are provided to initialize matrix P. The uniform distribution (UD) method is derived directly from [20] and gives the same probability for each edge, while the edge length (EL) method uses probabilities computed as a function of the lengths of the corresponding edges. According to the UD method, each edge is represented equally in the Cga population. Thus, we have

$$p^0_{i,j} = \begin{cases} 0, & \text{if } i = j \\ 1/2, & \text{otherwise.} \end{cases} \tag{2}$$

On the other hand, by using the EL method, a higher probability is assigned to the shorter edges according to the following equation:

$$p^0_{i,j} = \begin{cases} 0, & \text{if } i = j \\ \dfrac{\overline{L_i} - d(c_i, c_j)}{\overline{L_i} - \overline{l_i}}, & \text{otherwise} \end{cases} \tag{3}$$

where

$$\overline{L_i} = \max \{d(c_i, c_j) \colon j \in \{1, 2, \ldots, i-1\}\} \tag{4}$$

$$\overline{l_i} = \min \{d(c_i, c_j) \colon j \in \{1, 2, \ldots, i-1\}\}. \tag{5}$$

Generation of Feasible Tours: Traditional crossover operators cannot be applied to the TSP; randomly selecting parts of the parents' chromosomes and combining them into a new individual normally yields a tour in which some cities are not visited while some others are crossed more than once. *Ad hoc* crossover operators have, thus, been proposed that generate feasible tours [14]. As an example of such operators, *greedy crossover* [12] selects the first city of one parent, compares the cities leaving that city in both parents, and chooses the closest one to extend the tour. If this city has already appeared in the tour, the other city is chosen. If both cities have already appeared, an unselected city is taken randomly.

A greedy algorithm has also been designed to generate feasible tours from matrix P. A city c_a is randomly selected and inserted in the tour $\mathcal{V}$ as the starting city. Another city $c_b \notin \mathcal{V}$ is then chosen randomly. City c_b is inserted in $\mathcal{V}$ as the successor of c_a with probability $p_{a,b}$ [i.e., the probability associated to edge (c_a, c_b)]. Otherwise, c_b is discarded and the process is repeated by choosing another city not belonging to $\mathcal{V}$. Clearly, this process may fail to find the successor of some city $c_{\overline{a}}$. This happens when all the cities not already inserted in the current tour have been analyzed, but the probabilistic selection criterion failed to choose one of them. In this case, the city $c_{\overline{b}}$ successor of $c_{\overline{a}}$ is selected according to the following formula:

$$\overline{b} = \operatorname{argmax}\{p_{\overline{a},j} \colon c_j \in \{c_1, c_2, \ldots, c_k\} \setminus \mathcal{V}\}. \tag{6}$$

When (6) is satisfied by several cities [i.e., edges $(c_{\overline{a}}, c_j)$ have the same probability for different cities $c_j \notin \mathcal{V}$], the city that minimizes the distance $d(c_{\overline{a}}, c_j)$ is selected. The generation

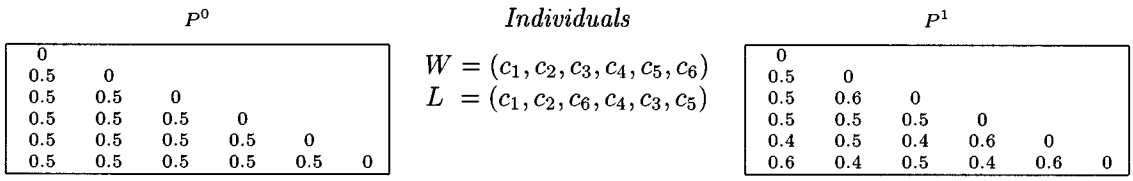

Fig. 2. Update step for a 6×6 probability matrix.

process ends when all the cities have been inserted in $\mathcal{V}$ and a feasible tour has been, thus, generated.

Such a generation operator has two main drawbacks.

1) It is expensive from a computational point of view. Its cost depends on the values stored in matrix P. The lower the probability values, the higher the computational cost since several edges are discarded before finding an edge which satisfies the probabilistic selection criterion. Moreover, as Cga execution progresses, very low probability values are assigned to most edges while only a few edge probabilities become significant. When the Cga converges, only two values of P are equal to one for each city, while all the other $k-2$ probability values are zero. Thus, the computational cost of our generation operator is lower at the beginning when high probabilities are associated with many edges, whereas it increases when the algorithm approaches convergence. By profiling the execution, we calculated the percentage of time spent in the `Generate()` subroutine. Depending on the TSP instance and Cga parameters, it ranges between 60%–70% of the total execution time.
2) It can generate individuals containing chromosomes that are not represented in the simulated population. In fact, when no city is selected as successor of $c_{\overline{a}}$ with the probabilistic selection criterion, the successor is chosen using (6). Particularly, when a few cities have to be inserted to complete the tour, all the probabilities checked by (6) might be equal to zero. Nevertheless, a city $c_{\overline{b}}$ is chosen although edge $(c_{\overline{a}}, c_{\overline{b}})$ is not present in any individual of the simulated population (i.e., $p_{\overline{a},\overline{b}} = 0$). Note that this behavior is also common to other TSP crossover operators and it is useful since it favors solution diversity by introducing mutations in the population.

Update of the Probability Matrix: The update protocol for the probability matrix P can be derived from that described by (1). Formally, given two individuals W and L so that W fitness is better than L, the probability values at step $k+1$ can be derived from that of step k as follows:

$$p_{i,j}^{k+1} = \begin{cases} p_{i,j}^{k} + \dfrac{1}{n}, & \text{if } (c_i, c_j) \vee (c_j, c_i) \in W \\ & \text{and } (c_i, c_j) \vee (c_j, c_i) \notin L \\ p_{i,j}^{k} - \dfrac{1}{n}, & \text{if } (c_i, c_j) \vee (c_j, c_i) \in L \\ & \text{and } (c_i, c_j) \vee (c_j, c_i) \notin W \\ p_{i,j}^{k}, & \text{otherwise} \end{cases} \tag{7}$$

where n is the size of the population. Fig. 2 shows a 6×6 probability matrix that was initialized according to the UD method and updated after the generation of individuals $W = (c_1, c_2, c_3, c_4, c_5, c_6)$ and $L = (c_1, c_2, c_6, c_4, c_3, c_5)$. In this example, we considered a population of ten individuals and, thus, probabilities associated with edges (c_3, c_2), (c_5, c_4), (c_6, c_1), and (c_6, c_5), which belong to W and not to L, were incremented by 0.1. On the other hand, probabilities for edges (c_5, c_1), (c_5, c_3), (c_6, c_2), and (c_6, c_4), which belong to L, but are not present in W, were decremented by 0.1.

End Condition: The Cga proposed in [20] ends when all probability values converge to zero or one and the probabilities themselves represent the final solution. Such a condition is rarely achieved in our case because the generation operator also allows edges to be inserted in a tour when their probabilities are very low. Since s individuals are generated at each Cga iteration and matrix P is updated by comparing their chromosomes (see Fig. 1), some edges may appear in the best of the generated s individuals although they do not belong to a TSP suboptimal solution. As a consequence, the probabilities values associated with these "bad" edges are increased, thus, jeopardizing the satisfaction of the termination condition that requires probabilities to be zero or one all at the same time. To minimize the effect of "wrong" increases of edge probabilities, the best individual generated in the current Cga iteration (`S[idx_best]`) is compared with the best individual found until now (`S_best`) and P updated accordingly. This modification favors the algorithm convergence and allows our Cga to reach the above end condition for small values of s. For large s, it is not sufficient since the probabilities associated with such "bad" edges might be increased for $s-1$ times and decreased only once at each Cga iteration. A supplementary end condition has been, thus, introduced, which limits the maximum number of generations occurring without an improvement of the best solution achieved (see Fig. 1). When such a limit is reached, execution is terminated and the best individual found is returned as the final solution.

B. Experimental Results

Table I reports the results of some tests conducted with our Cga on TSP instances `gr48`, a 48-city problem that has an optimal solution equal to `5046`, and `lin105`, a 105-city problem that has an optimal solution equal to `14 379`. These instances are available from TSPLIB [21], a library of TSPs.[2] The tests were carried out on a 200-MHz PentiumPro PC running Linux 2.2.12 by varying the method used for the initialization of the probability matrix (UD or EL), the size of the simulated population ($n = 500, 1000, 2000$ for `gr48` and $n = 1000, 2000, 4000$ for

[2]The TSPLIB collection is available at http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html.

TABLE I
TOUR LENGTH AND EXECUTION TIMES (IN SECONDS) OBTAINED RUNNING OUR Cga ON INSTANCES gr48 AND lin105

TSP Instance	n	s		UD			EL		
				Tour length	% Opt.	Time	Tour length	% Opt.	Time
gr48	500	2	bst	5055	0.18		5046	0	
			avg	5090	0.87	37	5088	0.85	33
		4	bst	5046	0		5046	0	
			avg	5081	0.70	231	5083	0.75	222
		8	bst	5046	0		5046	0	
			avg	5092	0.91	251	5068	0.43	248
		16	bst	5046	0		5046	0	
			avg	5084	0.76	275	5086	0.79	279
	1000	2	bst	5054	0.16		5046	0	
			avg	5108	1.22	85	5092	0.91	82
		4	bst	5055	0.18		5046	0	
			avg	5093	0.93	324	5085	0.77	318
		8	bst	5046	0		5046	0	
			avg	5082	0.71	399	5090	0.88	387
		16	bst	5046	0		5055	0.18	
			avg	5089	0.85	407	5080	0.67	398
	2000	2	bst	5046	0		5046	0	
			avg	5087	0.81	50	5095	0.97	49
		4	bst	5046	0		5046	0	
			avg	5087	0.81	272	5115	1.36	264
		8	bst	5046	0		5046	0	
			avg	5087	0.81	292	5087	0.81	275
		16	bst	5072	0.65		5046	0	
			avg	5086	0.79	352	5073	0.54	347
lin105	1000	2	bst	14497	0.82		14390	0.08	
			avg	14813	3.02	244	14634	1.77	231
		4	bst	14442	0.44		14379	0	
			avg	14711	2.31	1204	14580	1.40	1198
		8	bst	14379	0		14379	0	
			avg	14564	1.29	1567	14518	0.97	1394
		16	bst	14379	0		14379	0	
			avg	14474	0.66	1818	14486	0.75	2003
	2000	2	bst	14379	0		14390	0.08	
			avg	14695	2.2	446	14670	2.02	422
		4	bst	14401	0.15		14379	0	
			avg	14702	2.25	1544	14533	1.07	1497
		8	bst	14379	0		14379	0	
			avg	14491	0.78	1844	14493	0.80	1821
		16	bst	14379	0		14401	0.15	
			avg	14488	0.76	2203	14491	0.78	2089
	4000	2	bst	14459	0.56		14449	0.49	
			avg	14790	2.86	1571	14793	2.51	1529
		4	bst	14448	0.48		14401	0.15	
			avg	14658	1.94	3213	14579	1.39	3274
		8	bst	14379	0		14379	0	
			avg	14490	0.78	4931	14441	0.43	4692
		16	bst	14379	0		14379	0	
			avg	14458	0.55	6098	14422	0.30	5784

lin105), and the selection pressure ($s = 2, 4, 8, 16$). Each run was repeated ten times to obtain an average behavior. Table I reports the best (bst) and average (avg) tour length achieved with the ten executions, the distance of the solutions achieved from the global optimum (% Opt.), and the average execution time (Time).

As Table I shows, in 19 tests out of 24, our Cga found the optimal solution of the gr48 instance, while the global optimum of lin105 was found in 14 tests out of 24. Moreover, average solutions were also very close to the optimum. In all gr48 tests but one, we obtained solutions that, on average, differed by less than 1% from the optimum. The same holds for the lin105 tests if we consider the tests exploiting selection pressures to be higher than four. Average fitness values that differ for more than 3% from the optimum were obtained only in one case ($p = 1000, s = 2, UD$). The EL method for matrix initialization allowed the Cga to reach, in general, slightly better solutions in less time than the UD technique. Clearly, the EL initialization method allows the Cga to generate better individuals for the first generations. However, when simulated populations are large, as execution progresses the probability matrix approximately converges on the same solutions. Moreover, Table I shows that large populations and high selection pressures have to be simulated in order to find high quality average solutions. Increasing population sizes and selection pressures clearly results in a corresponding increase in the execution times.

To further validate our algorithm for the TSP, we compared its results with those obtained with an sGA. For this purpose, we used the TSP GA code contained in GAlib version 2.4.5, a C++ library of GAs developed by Wall.[3] Tests were carried out on the lin105 instance by fixing for both algorithms to 1000, the size of the population, and to 500 000, the number of evaluations of the fitness function performed. In addition, we used a selection pressure of ten for the Cga and a mutation probability of 0.5 for the sGA. Fig. 3 plots the length of the best tour found as a function of the number of the fitness function evaluations performed for a single run of the sGA and Cga implementations. As it can be seen, the macroscopic behavior of the two algorithms is sim-

[3]The GAlib library is available at http://lancet.mit.edu/ga/GAlib.html.

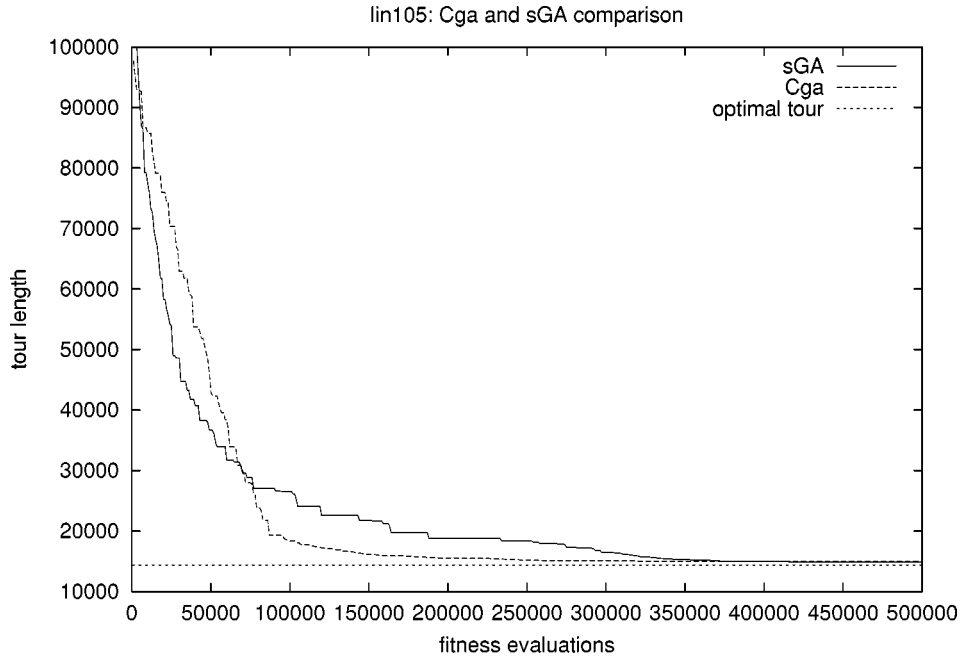

Fig. 3. Length of the best tour versus the number of fitness function evaluations performed by sGA and Cga algorithms on instance `lin105`.

ilar. However, calculating the average of 20 runs, our Cga was 1.9 times faster than the sGA in performing the same number of function evaluations. Moreover the Cga returned solutions that differ by 1.79% on average from the optimum, while the average distance from the optimum of the tours returned by the sGA was 4.36%.

III. Hybrid Approach Combining Genetics and the Lin–Kernighan Local Search

In Section II, we discussed the exploitation of a Cga to solve the TSP. Due to the huge population required and the corresponding increase in the number of generations needed to reach algorithm convergence, the Cga was found to be suitable only for small problem instances. On the other hand, it is known that local search heuristics that exploit domain-specific knowledge are very efficient for solving the TSP [3]. In particular, LK, the elegant algorithm of Lin and Kernighan [7], is the basis of most successful approaches proposed over the years for solving the TSP. The algorithm is quite complex and many enhancements to its original formulation have been proposed. In the following, we will give a simplified description of the algorithm. For a detailed explanation and a survey of the proposed enhancements and implementations, see [3] and [10].

The simplest way to view the overall LK algorithm is like a 3-Opt with the LK basic optimization step nested inside. All possible 3-Opt moves are applied systematically to the current *champion* (i.e., the best tour found so far) and the LK basic optimization step is applied to each one of the different tours built in this way. Whenever a locally optimal tour is found that is better than the current champion, it becomes the champion and the optimization process is restarted. The overall process terminates when all possible 3-Opt moves have been applied to the champion without yielding an improvement.

The LK basic optimization step exploits 2-Opt moves within a tabu search framework [8], [9]. Rather than applying single 2-Opt moves that reduce the length of the tour, it attempts to build a sequence of 2-Opt moves that globally yield an improvement, even if some of them may produce intermediate tours with larger costs than the initial one. About tabu conditions, the Lin and Kernighan original algorithm maintains two lists, one of *added* edges and one of *deleted* edges. A move is *tabu* if it attempts to delete an edge previously added (i.e., belonging to the added edge list) or add an edge previously deleted (i.e., belonging to the deleted edge list). The tabu search algorithmic framework, thus, limits the computational cost of the LK basic optimization step, which terminates when there are no more profitable nontabu moves for the current tour.

Computational efficiency is very important, since LK must be executed several times within an overall tour-finding schema in order to explore different regions of the search space, thus, improving the quality of the final solution. Among the most important tour-finding frameworks proposed to guide the LK search, we cite the *iterated LK* by Johnson and McGeoch [3] and the *chained local optimization* by Martin *et al.* [16], [17]. In iterated LK, an efficient implementation of LK is iteratively applied to different initial tours and the best of the tours selected. Chained local optimization is instead based on the idea of preserving part of the knowledge obtained with previous executions of LK. The locally optimal tour returned by LK is modified by applying a special type of 4-Opt move called a *double bridge* and LK is executed again on the new tour. An example of Martin *et al.*'s double-bridge move is shown in Fig. 4. It exchanges four edges in the tour for four other edges and it can perturb the original tour so that LK cannot undo, thus, avoiding the searching process being trapped in that local optimum.

Note that the acceptance of such a double bridge is decided on the basis of a SA schema that considers a probability associated with the difference in the length between the original and

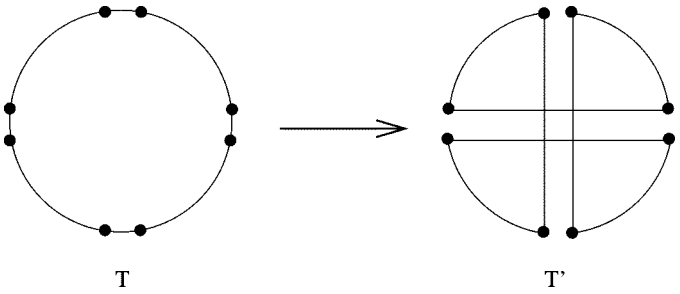

Fig. 4. Example of a double bridge.

the perturbed tour. Chained local optimization, thus, can be considered a hybrid algorithm that exploits SA to improve the exploration of the solution space. Analogously, we used our Cga to explore the more promising part of the TSP solution space. The Cga generates "good" initial solutions, which are then refined with LK. On the other hand, refined solutions are exploited to update the Cga probability matrix, thus, improving the simulated population in order to generate, as the execution progresses, better and better individuals that may lead LK to find tours with lower costs. Hereafter, we will call such hybrid algorithm Cga-LK.

Cga-LK exploits the efficient implementation of the LK heuristic available in the CONCORDE library by Applegate *et al.*[4] In particular, we used their chained-LK routine, which is a particular example of the Martin *et al.*chained local optimization, where random double-bridge moves are exploited and no SA is used [10].

Only slight modifications were needed to integrate the exploitation of CONCORDE chained-LK routine in the code reported in Fig. 1. We modified the method used to initialize the probability matrix. Since LK gives us an efficient way of generating near-optimal tours, we initially applied the LK routine to n randomly generated solutions (with n number of individuals of the simulated population) and increased by $1/n$ the probability associated with all the edges belonging to each one of the n optimized tours. In this way, we sensibly improve Cga behavior since the algorithm starts from a probability matrix that simulates a population of local optima. A further consequence of this is that Cga-LK also works effectively with a smaller population with respect to the pure Cga algorithm. Moreover, the selection pressure, which was very important for the pure Cga algorithm (see Section II-B), loses most of its importance since LK produces locally optimal tours. At each iteration i of Cga-LK, we, thus, generated a single individual s_i (from the probability matrix) and used it as a starting solution for the chained-LK routine, which produces tour $\overline{s_i}$. We then updated the probability matrix by comparing s_i with $\overline{s_i}$ as discussed in Section II-A.

A. Experimental Results

Cga-LK was tested on several medium/large TSP instances defined in TSPLIB [21]. Table II shows for each of the instances used as test cases, the TSPLIB name, the size, and the length of the optimal tour.

[4]The CONCORDE package is available for academic research use at http://www.keck.caam.rice.edu/concorde.

TABLE II
TSPLIB INSTANCES USED AS TEST CASES

Name	Cities	Optimal Solution
d198	198	15780
lin318	318	42029
pcb442	442	50778
att532	532	27686
gr666	666	294358
rat783	783	8806
pr1002	1002	259045
u2152	2152	64253
fl3795	3795	28772
fnl4461	4461	182566
frl5915	5915	565530

To demonstrate that genetically generated initial tours, allow the chained-LK routine to find higher quality solutions than nongenetically generated ones, we implemented two other algorithms: *random-LK* and *greedy-LK*, which both exploit the same CONCORDE chained-LK implementation. In the first one, LK initial tours are generated randomly; in the other one, a simple greedy algorithm is used for this purpose.

Table III shows the results obtained by running Cga-LK, random-LK, and greedy-LK on the same TSP instances. For each TSP instance and algorithm, Table III reports the best (bst) and average (avg) tour length as well as its distance from the global optimum (%), the best and the average of both the execution time (Time), and the number of iterations performed (Iter.). The tests were carried out on a 350-MHz PentiumIII PC running Linux 2.2.12. For the chained-LK routine, we used the CONCORDE default settings and we set to $i \cdot 5$, with i as the iteration index, the number of random double-bridge moves allowed within each execution of the chained-LK routine [10]. This setting was done to increase the depth of the local search as the execution progresses and more promising starting tours are generated by the Cga. The size of the simulated population was 64 for instances up to 442 cities, 128 for those up to 1002, and 256 for the others.

For the TSP instances d198, lin318, pcb442, att532, gr666, rat783, and pr1002, we ran each algorithm ten times. Execution was stopped as soon as the optimal tours were found. With instances u2152, fl3795, fnl4461, and frl5915, due to their high execution time, each execution was repeated five times. A limit on the execution time was fixed so that the execution was stopped when the optimal tour was found or the time limit was reached.

As shown in Table III, tests with instances d198, lin318, pcb442, and att532 gave similar results. All the three algorithms always found the optimal solutions of these TSP instances in a few seconds. They appeared substantially equivalent, showing the efficacy of the chained-LK algorithm exploited by all the three implementations.

Things changed when larger TSP instances were considered. The three algorithms always found the optimal solutions also on instances gr666, rat783, and pr1002, but the differences in the average execution times became noticeable. With instance gr666, Cga-LK found the optimal tour 1.39 and 1.49 times faster than random-LK and greedy-LK, respectively. For rat783, Cga-LK outperformed 2.84 times random-LK and 4.17 times greedy-LK. With the pr1002 instance, the Cga-LK

TABLE III
RESULTS OBTAINED BY RUNNING THE ALGORITHMS Cga-LK, RANDOM-LK, AND GREEDY-LK ON THE SAME TSP INSTANCES

TSP Inst.		Cga-LK			Random-LK			Greedy-LK		
		Iter.	Tour length (%)	Time	Iter.	Tour length (%)	Time	Iter.	Tour length (%)	Time
d198	bst	2	15780 (-)	0.2	2	15780 (-)	0.2	4	15780 (-)	0.5
	avg	10.7	15780 (-)	2.0	8.1	15780 (-)	1.5	7.4	15780 (-)	1.1
lin318	bst	16	42029 (-)	2.5	16	42029 (-)	2.5	15	42029 (-)	2.3
	avg	35.8	42029 (-)	12.1	32.4	42029 (-)	9.8	37.5	42029 (-)	12.8
pcb442	bst	34	50778 (-)	8.5	32	50778 (-)	7.2	42	50778 (-)	11
	avg	58.7	50778 (-)	21.7	84.1	50778 (-)	48.8	77.7	50778 (-)	36.8
att532	bst	57	27686 (-)	62	54	27686 (-)	51	41	27686 (-)	31
	avg	77	27686 (-)	112	78.9	27686 (-)	113	69.5	27686 (-)	92
gr666	bst	64	294358 (-)	140	63	294358 (-)	99	70	294358 (-)	125
	avg	124	294358 (-)	473	158	294358 (-)	659	165	294358 (-)	705
rat783	bst	97	8806 (-)	56	88	8806 (-)	38	143	8806 (-)	97
	avg	145	8806 (-)	111	263.8	8806 (-)	316	315	8806 (-)	463
pr1002	bst	103	259045 (-)	104	132	259045 (-)	141	204	259045 (-)	314
	avg	125	259045 (-)	145	398.3	259045 (-)	1294	377	259045 (-)	1118
u2152	bst	332	64253 (-)	918	1064	64308 (0.086)	5000	1058	64322 (0.107)	5000
	avg	495	64253 (-)	1772	1065	64356.0 (0.160)	5000	1059	64352.9 (0.155)	5000
fl3795	bst	413	28772 (-)	3897	999	28772 (-)	13444	751	28772 (-)	7706
	avg	485	28772 (-)	5033	1037	28774.4 (0.008)	14844	977	28772.7 (0.002)	13491
fnl4461	bst	1358	182566 (-)	23867	1906	182684 (0.065)	35000	1904	182705 (0.076)	35000
	avg	1679	182578.4 (0.007)	33887	1910	182722.4 (0.086)	35000	1906	182719.5 (0.084)	35000
frl5915	bst	1444	565530 (-)	30935	1958	566052 (0.092)	35000	1947	566079 (0.097)	35000
	avg	1546	565554.0 (0.004)	34187	1965	566131.4 (0.106)	35000	1950	566159.7 (0.111)	35000

performance improvement was impressive: 8.92 and 7.71 over random-LK and greedy-LK, respectively.

With instance u2152, Cga-LK found the optimal tour with each execution, while neither random-LK nor greedy-LK ever found it. With instance fl3795, all the three algorithms found the optimal tour, but Cga-LK succeeded in all the five executions, while random-LK found the optimum once and greedy-LK twice. Cga-LK found the optimal tour also with instances fnl4461 and frl5915, while the optimal solution were only approximated by random-LK and greedy-LK. Moreover, when the optimal tour was not found and execution stopped because of the limit in the execution time, the quality of the solutions returned by Cga-LK was always higher than those carried out by the other two methods. This happened even if Cga-LK, in a fixed time limit, performs fewer LK searches than random-LK and greedy-LK due to its higher computational cost in building starting solutions. As an example, with instance frl5915, Cga-LK on average performed 1546 iterations and returned a final tour difference of only 0.004% from the optimal one, while the other two algorithms, within the same time limit, executed about 400 more chained-LK searches, but returned solutions with errors that were more than 25 times higher.

Cga-LK was tested also on usa13509, a large TSP instance that represents the Euclidean distance among the 13 509 cities with a population of at least 500 in the continental U.S. With this instance, Cga-LK was performed just once due to the high execution time required. We stopped the test after 2800 iterations (about 137 hours) and achieved a solution with length equal to 19 991 585, which was a percentage difference of only 0.043% from the optimal tour length of 19 982 859 found in 1998 by Applegate *et al.* They used a parallel implementation of an exact method based on linear programming [2]. The authors estimated that to carry out the optimal solution of the instance usa13509 on a sequential machine would take approximately ten years.

Fig. 5(a) plots the distance (in percentage) from the optimal tour of the solutions returned by the chained-LK routine as a function of the iteration index. In the plot, the dots represent the solution returned by the current execution of chained LK, the solid line plots the distance of the best solution, while the dotted line plots the average distance from the optimum of the last 25 solutions. As we can see, the distance from the optimum of both average and best solutions decreases as execution progresses. For the same test, Fig. 5(b) plots the distance (in percentage) from the optimal tour of the solutions generated by our Cga and used to start the chained-LK routine. The dots represent the distance from the optimum of the current Cga solution, while the solid line plots the average distance of the last 25 solutions generated with genetics. In this case, we can also see that the average quality of the solutions tends to improve as execution progresses, thus, demonstrating that the symbiosis between the two heuristics works effectively.

IV. CONCLUSION

The main contributions of this paper were to show that the Cga can be successfully extended to deal with "difficult" order-k optimization problems such as the TSP and that its peculiarities in terms of both implementation easiness and memory requirements make the Cga particularly suitable to design hybrid algorithms combining in a good symbiotic behavior genetic and local search heuristics.

Our Cga for the TSP was evaluated on small problem instances. The results achieved were satisfactory if compared to those obtained with an sGA. Memory requirements are strongly reduced compared to traditional GA implementations and very large populations can be used in order to improve the quality of the solutions found. In addition, the Cga behavior depends on just a few parameters (namely, the size of the population and the selection pressure), thus, making the Cga easier to tune than a GA.

The above characteristics simplify the design of Cga-LK, a hybrid heuristic algorithm that combines our Cga for the TSP with the LK local search heuristics designed by Lin and Kernighan. In Cga-LK, the Cga is used as an overall

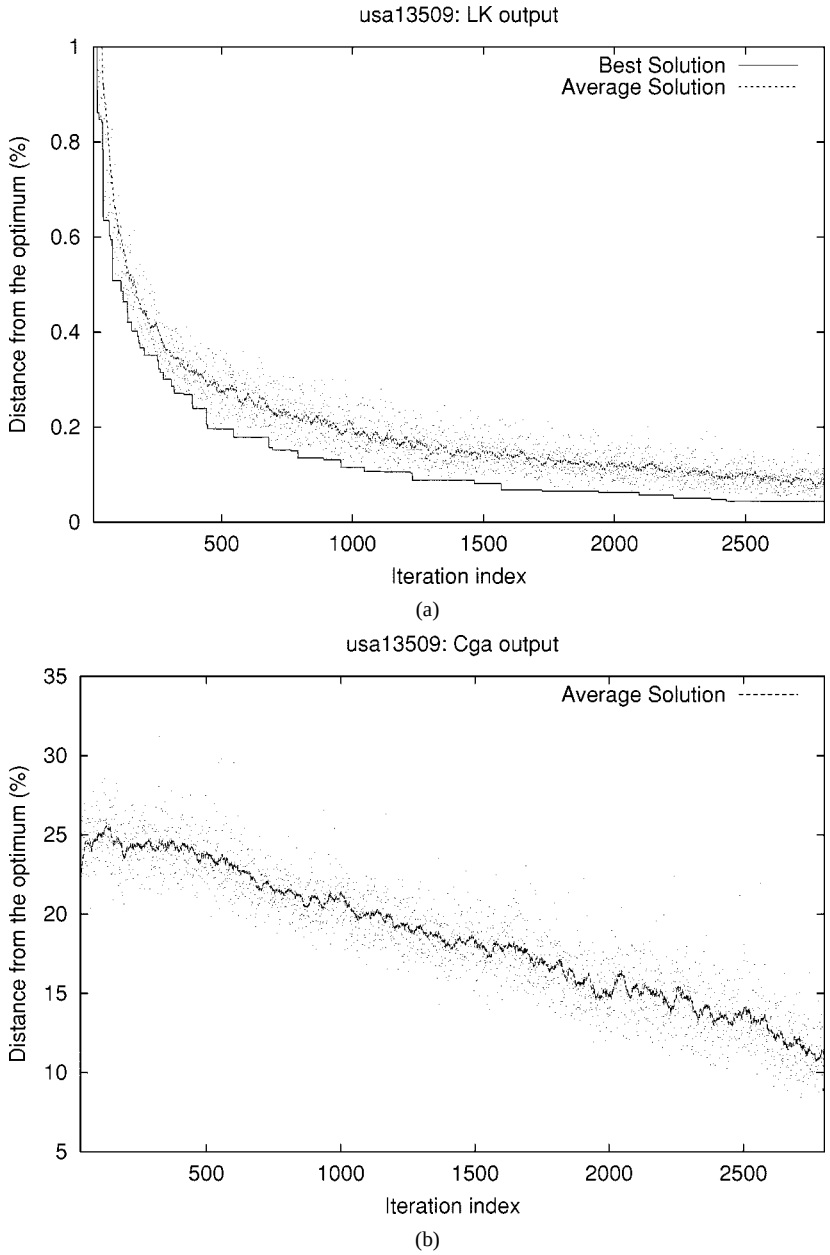

Fig. 5. Distance from the optimum of the tours returned by (a) the chained-LK routine and (b) the Cga part of our hybrid algorithm on the TSP instance `usa13509` as a function of the iteration index.

tour-finding schema to generate "good" LK starting tours. Local optima returned by LK are in turn exploited to improve, as execution progresses, the simulated population within the Cga. Genetics is, thus, used to incorporate into the algorithm part of the "knowledge" obtained in the previous LK runs.

Cga-LK was tested with several TSPLIB instances with up to 13 509 cities and we demonstrated the efficacy of the symbiosis between genetics and LK by comparing the results achieved with those obtained with two other non-GAs exploiting the same LK implementation on starting tours generated from scratch. On large TSP instances, Cga-LK gets better performances than the other two algorithms in terms of execution time or quality of the solutions found or both at the same time. At our best knowledge, the computational results obtained with Cga-LK are the best published regarding genetic approaches to the TSP.

As future work, we plan to investigate extensions of the Cga for other complex optimization problems. In particular, we are interested in the multi-field-programmable gate-array (FPGA) partitioning problem [27]. We expect that the huge memory requirements that constrain the solution of complex multi-FPGA partitioning problems via GAs can be weakened by exploiting the Cga.

ACKNOWLEDGMENT

The authors would like to thank D. Applegate, R. Bixby, V. Chvatal, and W. Cook, the authors of the CONCORDE library exploited by Cga-LK, and M. Wall who implemented the GAlib code they used to validate their Cga for the TSP. They would also like to thank D. B. Fogel and the anonymous referees for

their useful suggestions that allowed for the improvement of this paper.

REFERENCES

[1] G. Reinelt, *The Traveling Salesman: Computational Solutions for TSP Applications*. Berlin, Germany: Springer-Verlag, 1994.

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, "On the solution of traveling salesman problems," in *Documenta Mathematica: Proc. Int. Cogr. Mathematicians*, vol. 3, 1998, pp. 645–656.

[3] D. S. Johnson and L. A. McGeoch, "Local search in combinatorial optimization," in *The Traveling Salesman Problem: A Case Study in Local Optimization*. New York: Wiley, 1996.

[4] M. Padberg and G. Rinaldi, "Optimization of a 532-city symmetric genetic traveling salesman problem by branch and cut," *Oper. Res. Lett.*, vol. 6, no. 1, pp. 1–7, 1987.

[5] G. A. Croes, "A method for solving traveling salesman problems," *Oper. Res.*, vol. 6, no. 6, pp. 791–812, 1958.

[6] S. Lin, "Computer solution of the traveling salesman problem," *Bell Syst. Tech. J.*, vol. 44, pp. 2245–2269, 1965.

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

[8] F. Glover, "Tabu search—Part I," *ORSA J. Comput.*, vol. 1, no. 3, pp. 190–206, 1989.

[9] ——, "Tabu search—Part II," *ORSA J. Comput.*, vol. 2, no. 1, pp. 4–32, 1990.

[10] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. (1999) *Finding tours in the TSP* [Online]. Available: http://www.caam.rice.edu/keck/reports/lk_report.ps

[11] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[12] J. Grefenstette, R. Gopal, B. Rosimaita, and D. van Gucht, "Genetic algorithms for the traveling salesman problem," in *Proc. Int. Conf. Genetics Algorithms and Their Applications*, July 1985, pp. 160–168.

[13] H. C. Braun, "On solving traveling salesman problems by genetic algorithm," in *Parallel Problem Solving from Nature*, H. P. Schwefel and R. Männer, Eds. Berlin, Germany: Springer-Verlag, 1991, vol. 496, Lecture Notes in Computer Science, pp. 129–133.

[14] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd ed. Berlin, Germany: Springer-Verlag, 1996.

[15] G. Tao and Z. Michalewicz, "Inver-over operator for the tsp," in *Parallel Problem Solving from Nature, PPSN V*. Berlin, Germany: Springer-Verlag, 1998, vol. 1498, Lecture Notes in Computer Science, pp. 803–812.

[16] O. Martin, S. W. Otto, and E.W. Felten, "Large step Markov chain for the traveling salesman," *J. Complex Syst.*, vol. 5, no. 3, p. 299, 1991.

[17] O. Martin and S. W. Otto, "Combining simulated annealing with local search heuristic," *Ann. Oper. Res.*, vol. 63, pp. 57–75, 1996.

[18] P. Merz and B. Freisleben, "Genetic local search for the TSP: New results," in *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*. Piscataway, NJ: IEEE Press, 1997, pp. 159–163.

[19] M. Gorges-Schleuter, "Asparagos96 and the travelling salesman problem," in *Proceedings of the Fourth International Conference on Evolutionary Computation*, T. Bäck, Ed. Piscataway, NJ: IEEE Press, 1997, pp. 171–174.

[20] G. Harik, F. Lobo, and D. Goldberg, "The compact genetic algorithm," *IEEE Trans. Evol. Comput.*, vol. 3, pp. 287–297, Nov. 1999.

[21] G. Reinelt, "TSPLIB—A traveling salesman problem library," *ORSA J. Comput.*, vol. 3, no. 4, pp. 376–384, 1991.

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

[23] G. Harik, D. Goldberg, and B. Miller, "The gamblers ruin problem, genetic algorithms, and the sizing of populations," in *Proceedings of the Fourth International Conference on Evolutionary Computation*, T. Bäck, Ed. Piscataway, NJ: IEEE Press, 1997, pp. 7–12.

[24] G. Syswerda, "Simulated crossover in genetic algorithms," in *Foundation of Genetic Algorithms 2*, L. D. Whitley, Ed. San Mateo, CA: Morgan Kaufmann, 1993, pp. 239–255.

[25] S. Baluja, "Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning," Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-94-163, 1994.

[26] D. Thierens and D. Goldberg, "Mixing in genetic algorithms," in *Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest, Ed. San Mateo, CA: Morgan Kaufmann, 1993, pp. 38–45.

[27] S. Hauck, "The roles of FPGAs in re-programmable systems," *Proc. IEEE*, vol. 86, pp. 615–638, Apr. 1998.