Consiglio Nazionale delle Ricerche

A Hybrid Approach for the TSP Combining Genetics and the Lin-Kerninghan Local Search

Ranieri Baraglia, Jose Ignacio Hidalgo
Raffaele Perego

Technical Report
CNUCE-B4-2000-007

CNUCE
Pisa

A Hybrid approach for the TSP combining genetics and the Lin-Kerninghan local search

Ranieri Baraglia, Raffaele Perego and Jose Ignacio Hidalgo

Istituto CNUCE, Consiglio Nazionale delle Ricerche (CNR)

Via V. Alfieri, 1, Ghezzano, Pisa, Italy

e-mail: {ranieri.baraglia, raffaele.perego}@cnuce.cnr.it hidalgo@dacya.ucm.es

Technical Report: #7/2000

2nd May 2000

Abstract

The combination of genetic and local search heuristics has been shown to be an effective approach to solving the Traveling Salesman Problem. 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 local search (LK). Local optima found by LK 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 13509 cities show the efficacy of the symbiosis between the two heuristics.

Keywords: compact genetic algorithms, Lin-Kernighan, Hybrid GA, TSP

Contents

1 Introduction **4**

2 Genetic algorithms **8**

2.1 The Compact Genetic Algorithm 9

2.2 A *Cga* for the TSP 12

2.3 Experimental Results 18

3 A hybrid approach combining genetics and the Lin-Kernighan local search **21**

3.1 Experimental Results 23

4 Conclusions **29**

List of Figures

2.1 Pseudo-code of our *Cga* for the TSP. 13

2.2 Update step for a 6×6 probability matrix. 16

3.1 Pseudo-code of our *Cga-KL* for the TSP. 24

3.2 Distance from the optimum of the tours returned by (a) the *Chained LK* routine, (b) the *Cga* part of our hybrid algorithm on the TSP instance `usa13509`, as a function of the iteration index. 26

List of Tables

I Tour length and execution times (in seconds) obtained running our *Cga* on instances `gr48` and `lin105`. 19

I TSPLIB instances used as test cases. 23

II Comparison of results obtained by running the algorithms *Cga-LK*, *Random-LK*, and *Greedy-LK* on the same TSP instances. 26

Chapter 1

Introduction

The Traveling Salesman Problem (TSP) is probably the most famous combinatorial optimization problem. In the TSP, given a finite set of *cities* $C = \{c_1, c_2, ..., c_k\}$ and a distance $d(c_i, c_j), i \neq j$, between each pair, the shortest tour which visits all the cities once and returns to the starting point has to be found. Because of its simplicity and applicability to many fields, TSP has often served as a testing ground for many exact and heuristic optimization algorithms [24, 1, 13]. Highly optimized exact algorithms based on the *Branch & Cut* method [22, 1] have been proposed which 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* [6], *3-Opt* [15], and LK [16], are surprisingly very effective for the TSP. In particular LK, which essentially uses *2-Opt* and *3-Opt* moves from within a *tabu search* algorithmic framework, is considered to be 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 which take just a few seconds to compute a high-quality solution for problems with hundreds of cities [13, 2]. Computational efficiency is very important since LK, as with all local search algorithms, must be executed several times with different random seeds and starting solutions 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) [14] and genetic algorithms (GA) [9, 4] perform quite poorly with this particular problem. They require high execution times for solutions whose quality is not comparable with those achieved in much less time by their domain-specific local search counterparts.

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 TSP problems. The problem-independent part of the hybrid algorithm drives the breadth-first exploration of the search space thus focusing on the global optimization task, while the local search algorithm allows a depth-first search of the subregions of the solution space, to be efficiently performed.

Martin and Otto proposed the *Chained local optimization* algorithm, where a special type of *4-opt* moves are used under the control of a SA schema to escape from the local optima found with LK [18, 17].

Freisleben and Merz designed genetic operators to search the space of the local optima determined with LK [19]. In particular they used a specific crossover operator which preserves the edges found in both the parents. To our knowledge, Freisleben and Merz's

TSP results reported in [19] are currently the best obtained using an algorithm which combine genetics with a local search.

Martina Georges-Schleuter instead experimented with the exploitation of simple k-Opt moves within her *Asparagos96* parallel genetic algorithm [8]. She concludes 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 mimics its existence and stores the information describing the population in a triangular matrix. The idea behind this algorithm arose from the study of the compact genetic algorithm proposed by R. Harik and others [11]. 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 the attention on the *symmetric* TSP, and tested our algorithm on `TSPLIB` [23] instances with up to 13509 cities.

The paper is organized as follows. Chapter 2 briefly introduces the genetic approach and describes the compact genetic algorithm exploited by our proposal. Some results obtained on small TSP instances are also presented and discussed. Our hybrid approach which combines the compact genetic algorithm and the Lin-Kernighan local search is

detailed in Chapter 3. Section 3.1 presents the results achieved with our implementation on TSP instances ranging from 198 to 13509 cities. The results are also compared with those of other algorithms applied to the same problem instances. Finally, Chapter 4 outlines some conclusions.

Chapter 2

Genetic algorithms

Genetic Algorithms (GAs) [12, 21, 20] are stochastic optimization heuristics in which searches in solution space are carried out by imitating the population genetics stated in Darwin's theory of evolution. Selection, crossover and mutation operators, directly derived from natural evolution mechanisms are applied to a population of solutions, thus favoring the birth and survival of the best solutions. GAs have been successfully applied to many NP-hard combinatorial optimization problems, in several application fields such as business, engineering, and science.

In order to apply GAs to a problem, a genetic representation must be found of each individual (*chromosome*) that constitutes a solution to the problem. Then, we need to create an initial population, to define a cost function to measure the *fitness* of each solution, and to design the genetic operators that will allow us to produce a new population of solutions from a previous one. By iteratively applying the genetic operators to the current population, the fitness of the best individuals in the population converges to local optima. The GA end condition may be to reach a maximum number of generated populations, after which the algorithm is forced to stop, or the algorithm converges at

stable average fitness values.

The genetic approach has several advantages which make GAs usable and effective. Firstly, GAs do not deal directly with problem solutions but with their genetic representation thus making their implementation independent from the problem in question. Moreover, they do not treat individuals but rather populations, thus increasing the probability of finding good solutions. Finally, GAs use probabilistic methods to generate new populations of solutions, thus decreasing the probability of being trapped in "bad" local optima. On the other hand, GAs do not guarantee that global optima will be achieved and their efficacy very much depends on many parameters whose fixing depends on the problem considered. The size of the population is particularly important. The larger the population, the greater the possibility of profitably exploring the solution space, thus reaching good solutions. Increasing the population clearly results in a large increase in the GA computational cost which can be addressed by exploiting parallelism according to a few well-known models [7, 5].

2.1 The Compact Genetic Algorithm

The *Cga* does not manage a population of solutions but only mimics its existence [11]. 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 [10]. Other concepts that inspired the *Cga* proposal were *Bit-based Simulated Crossover* (BSC) [25] and *Population-Based Incremental Learning* (PBIL) [3]. 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 which has 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 can be updated accordingly 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:

$$p_i^{k+1} = \begin{cases} p_i^k + 1/n & \text{if } w_i = 1 \wedge l_i = 0 \\ p_i^k - 1/n & \text{if } w_i = 0 \wedge l_i = 1 \\ p_i^k & \text{if } w_i = l_i \end{cases} \qquad (2.1)$$

where n is the dimension 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 0 or 1. At this point vector p itself represents the final solution.

Let us see a simple example for 6 genes and a population of 10 individuals. The initial values of the population would be: $P^0 = [0.5 \;\; 0.5 \;\; 0.5 \;\; 0.5 \;\; 0.5 \;\; 0.5]$. Suppose that we generate the next two individuals: $Individual1 = 0\,1\,0\,1\,0\,0$ and $Individual2 = 0\,1\,0\,0\,1\,0$, and also that their fitness values are $FF_1 = 1.7$ and $FF_2 = 2.3$. So $W = Individual2$ and

$L = Individual1$ i.e. $L = 0\ 1\ 0\ 0\ 1\ 0$ and $W = 0\ 1\ 0\ 1\ 0\ 0$

Now we can calculate the new values of the probabilities P^1: $p_1^1 = p_1^0$ because $w_1 = l_1$. For the same reason $p_2^1 = p_2^0$, $p_3^1 = p_3^0$, and $p_6^1 = p_6^0$. However, $p_4^1 = p_4^0 + 1/10 = 0.6$,because $w_4 = 1$ and $l_4 = 0$. Moreover, $p_5^1 = p_5^0 - 1/10 = 0.4$,because $w_4 = 0$ and $l_4 = 1$. So the probability vector for generation 1 would be: $P^1 = [0.5\ \ 0.5\ \ 0.5\ \ 0.6\ \ 0.4\ \ 0.5]$.

The algorithm will continue working in this way until all of the values of P will be 0 or 1. Supouse that after 20 generations the algorithm converges and the value of the probabilities are: $P^{20} = [0\ 1\ 1\ 1\ 0\ 0]$. Then, the solution to our problem is that represented by the chromosome 0 1 1 1 0 0.

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 wrongly modified.

When applied to order-one problems a *Cga* is approximatively equivalent to a simple GA with uniform crossover: it achieves solutions of comparable quality with approximatively 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)** generate at each iteration 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; **(3)** compare W with the other $s - 1$ representations and update the probability vector accordingly. 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 [11].

Although the *Cga* mimics the order-one behavior of a GA with uniform crossover, it was not proposed as an alternative algorithm. According to the 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 considere as difficul. 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 requires $\log_2 n * l$ bits with respect to the $n * l$ bits needed by a classic GA. Larger population dimensions can be exploited without significantly increasing memory requirements.

2.2 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 [20], 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 dimension, our *Cga* thus requires $(k^2/2) \cdot \log_2 n$ bits to represent the population with respect to the $\log_2 k \cdot k \cdot n$ bits required by a classical *GA*. Figure 2.1 shows the pseudo-code of our *Cga*. Its main functions are discussed in the following.

```
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
```

Figure 2.1: Pseudo-code of our *Cga* for the TSP.

Initialization of the probability matrix

Two different methods are provided to initialize matrix P. The Uniform Distribution (UD) method 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 equally represented in the *Cga* population. Thus we have:

$$p^0_{i,j} = \begin{cases} 0 & \text{if } i = j \\ 1/2 & \text{otherwise} \end{cases} \tag{2.2}$$

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

$$p^0_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \frac{\overline{L_i} - d(c_i, c_j)}{\overline{L_i} - \overline{l_i}} & \text{otherwise} \end{cases} \tag{2.3}$$

where

$$\overline{L_i} = \max\{d(c_i, c_j) : j \in \{1, 2, \ldots, i-1\}\} \tag{2.4}$$

$$\overline{l_i} = \min\{d(c_i, c_j) : j \in \{1, 2, \ldots, i-1\}\}. \tag{2.5}$$

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 which generate feasible tours [20]. As an example of such operators, *Greedy Crossover* [9] 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, a non-selected city is randomly taken.

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 starting city. Another city

$c_b \notin \mathcal{V}$ is then randomly chosen. City c_b is inserted in $\mathcal{V}$ as 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:

$$\overline{b} = \operatorname{argmax}\{p_{\overline{a},j} : c_j \in \{c_1, c_2, \ldots, c_k\} \setminus \mathcal{V}\} \tag{2.6}$$

When Equation 2.6 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 which minimizes the distance $d(c_{\overline{a}}, c_j)$ is selected. The generation 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 clearly 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 there is a reasonable increase when the algorithm approaches convergence. We calculated that, depending on the TSP instance and *Cga* parameters, the percentage of time spent in the `Generate()` subroutine ranges between 60 and 70 per cent 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 Equation 2.6. Particularly when a few cities have to be inserted to complete the tour, all the probabilities checked by Equation 2.6 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 [9], and it is useful since it favors solution diversity by introducing *mutations* in the population.

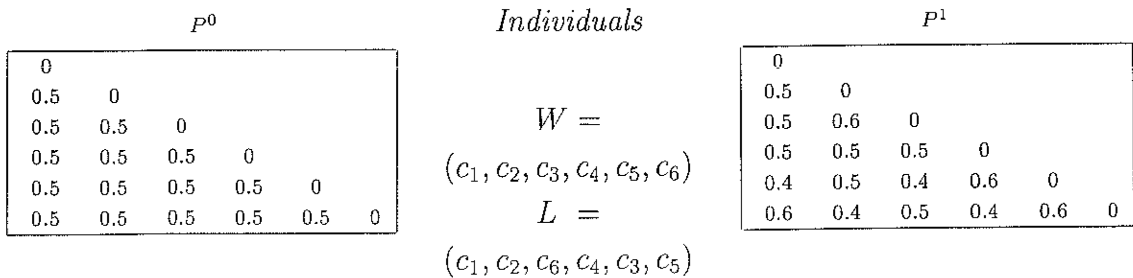


Figure 2.2: Update step for a 6×6 probability matrix.

Update of the probability matrix

The update protocol for the probability matrix P can be easily derived from that described by Equation 2.1. 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 follow:

$$p_{i,j}^{k+1} = \begin{cases} p_{i,j}^{k} + \frac{1}{n} & \text{if } (c_i, c_j) \vee (c_j, c_i) \in edges(W) \text{ and } (c_i, c_j) \vee (c_j, c_i) \notin edges(L) \\ p_{i,j}^{k} - \frac{1}{n} & \text{if } (c_i, c_j) \vee (c_j, c_i) \in edges(L) \text{ and } (c_i, c_j) \vee (c_j, c_i) \notin edges(W) \\ p_{i,j}^{k} & \text{otherwise} \end{cases} \tag{2.7}$$

where n is the dimension of the population. Figure 2.2 shows a 6×6 probability matrix which 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 10 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 $edges(W)$ and not to $edges(L)$ 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 $edges(L)$ but are not present in tour W, were decremented by 0.1.

End Condition

The *Cga* proposed in [11] ends when all probability values converge to 0 or 1, 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 (see chapter 2.2). Since s individuals are generated at each *Cga* iteration, and matrix P is updated by comparing their *chromosomes* (see Figure 2.1), some edges may appear in the best of the generated s individuals although

they do not belong to a TSP sub-optimal 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 Figure 2.1). When such a limit is reached, execution is terminated and the best individual found is returned as the final solution.

2.3 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 **14379**. These instances are available from TSPLIB [23], a library of traveling salesman problems [1]. The tests were carried out on a 200MHz PentiumPro PC running Linux 2.20.12, by varying the method used for the initialization of the probability matrix (UD or EL), the dimension of the simulated population ($n = 500, 1000, 2000$ for **gr48**, and $n = 1000, 2000, 4000$ for **lin105**), and the

[1] TSPLIB collect.:`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 Tour length	UD % Opt.	UD Time	EL Tour length	EL % Opt.	EL 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

selection pressure ($s = 2, 4, 8, 16$). Each run was repeated 10 times to obtain an average behavior. The Table reports the best (*bst*) and average (*avg*) tour length achieved with the 10 executions. The distance of the solutions achieved from the global optimum (% Opt.) as well as the average execution time (*Time*) are reported in the Table.

As the Table 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 the **gr48** tests but one we obtained solutions which, on average, differed for 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 4. Average fitness values which 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 than the UD technique. Clearly the EL initialization method allows the *Cga* to generate better individuals for the first generations. However, since simulated populations are large, as execution progresses the probability matrix converges at about the same solutions. Moreover the Table shows that large populations and high selection pressures have to be simulated in order to find high quality average solutions. Increasing population dimensions and selection pressures clearly results in a corresponding increase in the execution times.

Chapter 3

A hybrid approach combining genetics and the Lin-Kernighan local search

In the previous chapter 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 which exploit problem domain specific knowledge are very efficient for solving the TSP [13]. In particular LK, the elegant algorithm of Lin and Kernighan [16], is the basis of most successful approaches proposed over the years for solving the TSP. Although 27 years "old", LK is still considered to be the best improvement heuristic for the TSP, since it produces high quality solutions on a wide variety of TSP instances. A lot of variations of the LK algorithm have been proposed (see [13, 2] for a survey and a complete bibliography). Among the most important, we cite the *Iterated LK* by Johnson and McGeoch [13], and the *Chained local optimization* by Martin, Otto and Felten [18, 17]. In *Iterated LK*, an efficient implementation of LK is iteratively applied to different initial tours and the best

of the tours selected, whereas *Chained local optimization* applies a special type of *4-opt* moves (called *double bridge*) to *kick* the current tour found by LK before reapplying LK. Note that the acceptance of the *kicked* tours is decided on the basis of a *Simulated Annealing* schema which considers a probability associated with the difference in the length between the original and the *kicked* tour. *Chained local optimization* can be thus considered a hybrid algorithm which exploits *Simulated Annealing* 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 execution progresses, better and better individuals which 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, Bixby, Chvatal, and Cook [1]. In particular we used their *Chained LK* routine, which is a particular example of the Martin, Otto, and Feldman *Chained local optimization*, where *kicks* are random *double-bridge* moves and no *Simulated Annealing* is used [2].

As we can see from the pseudo-code reported in Figure 3.1, only slight modifications were needed to integrate the exploitation of CONCORDE *Chained LK* routine in the code reported in Figure 2.1. 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

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

Table I: 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

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 which 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 chapter 2.3), looses 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 chapter 2.2.

3.1 Experimental Results

Cga-LK was tested on several medium/large TSP instances defined in TSPLIB [23]. Table 3 shows for each of the instances used as test cases, the TSPLIB name, the size, and

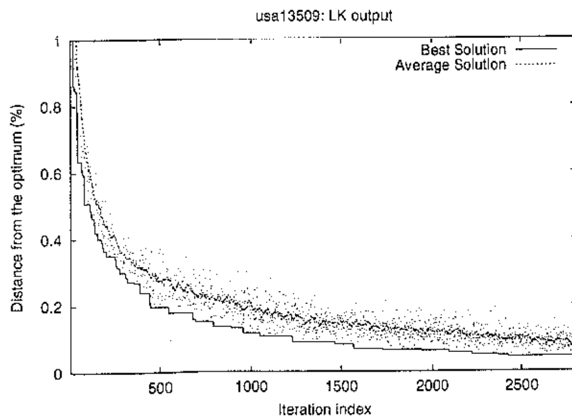
of the three algorithms on the same TSP instances, we were more able to understand if the quality of the solutions was only due to the established efficacy of LK. This is where other proposals fail since the contribution of genetics to the results is not shown.

Table II reports the results of such a comparison. In particular, for each TSP instance and algorithm, it reports: the best (*bst*) and average (*avg*) tour length as well as its distance from the global optimum (%), and the best and the average of both the execution time and the number of iterations performed. The tests were carried out on a 350MHz PentiumIII PC running Linux 2.2.12. We used for the *Chained LK* routine the `CONCORDE` default settings, and we set to $i \cdot 5$, with i the iteration index, the number of random *double-bridge* kicks allowed within *Chained LK* [2]. The dimension 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 one of the three algorithms 10 times. Execution was stopped as soon as the optimal tours were found, and best and average execution times were reported in the Table. With instances `u2152`, `fl3795`, `fnl4461`, and `frl5915`, each execution was instead repeated 5 times, and a limit on the execution time was fixed so that the execution was stopped when the optimal tour for the specific TSP problem was found or the time limit reached.

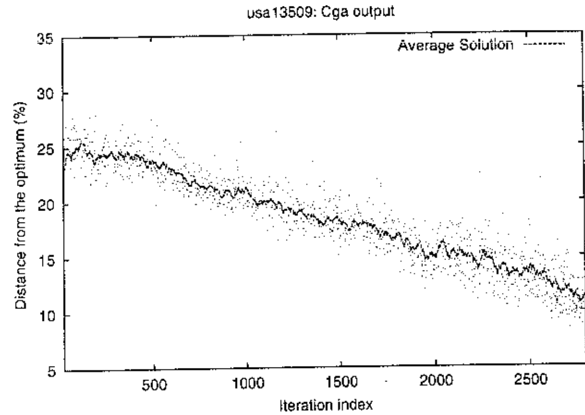
As we can see from the Table, the tests with instances `d198`, `lin318`, `pcb442`, and `att532` gave contrasting results. All the three algorithms always found the optimal solutions of these TSP instances in a few seconds. In some cases one of the three algorithms was slightly faster than another, but the three algorithms appeared substantially equivalent. The *Chained LK* algorithm exploited within the three implementations was very effective almost independently from the method used to generate the starting solutions.

Table II: Comparison of 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
rl5915	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



(a)



(b)

Figure 3.2: Distance from the optimum of the tours returned by (a) the *Chained LK* routine, (b) the *Cga* part of our hybrid algorithm on the TSP instance usa13509, as a function of the iteration index.

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 sensible. With instance gr666, *Cga-LK* resulted 1.39 and 1.49 times faster in finding the optimal tour 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* 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 5 executions, while *Random-LK* found the optimum once, and *Greedy-LK* twice. Differently from the other two algorithms, *Cga-LK* found the optimal tours also with instances fnl4461, and frl5915. 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 one of the other two methods. This holds although *Cga-LK* in a fixed time limit performs fewer LK searches than *Random-LK* and *Greedy-LK* due to the 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 which were more than 25 times higher.

Cga-LK was experimented also on usa13509, a large TSP instance which represents the Euclidean distance among the 13509 cities with a population of at least 500 in the

continental US. With this instance *Cga-LK* was performed just one 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 19991585, which was a percentage difference of only 0.043% from the optimal tour length of 19982859 found in 1998 by Applegate, Bixby, Chvatal, and Cook. They used a parallel implementation of an exact method based on linear programming [1]. The authors estimated that to carry out the optimal solution of the instance `usa13509` on a sequential machine would take approximately 10 years.

Figure 3.2.(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. Figure 3.2.(b) instead plots for the same test, the distance (in percentage) from the optimal tour of the solutions generated by our *Cga*, and used to start the *Chained LK* routine. As before, 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.

Chapter 4

Conclusions

In this paper we proposed *Cga-LK*, a new hybrid heuristic algorithm for the TSP. It combines an original *Cga* with an efficient implementation of the well-known LK local search heuristics designed by Lin and Kernighan. In *Cga-LK*, the *Cga* is used to generate what we hope will be "good" tours for starting LK. Local optima returned by LK are in turn exploited to improve, as execution progresses, the population simulated within the *Cga*. Genetics is thus used to incorporate into the algorithm, part of the "knowledge" obtained in the previous LK runs.

The evaluation of the proposed approach was done in two steps. Firstly, our pure *Cga* was evaluated using two TSPLIB instances with 48 and 105 cities, respectively. The results achieved with both the instances were satisfactory. In most of the tests our *Cga* found the optimal tour. Average solutions were also very close to the optimum. However, the experiments showed that large populations and high selection pressures have to be also exploited on small TSP instances in order to find average solutions of an acceptable quality. Increasing population dimensions and selection pressures clearly resulted in an increase in the *Cga* execution times which makes the pure *Cga* suitable only for dealing

Bibliography

[1] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the solution of traveling salesman problems. In *Documenta Mathematica*, volume Extra Volume ICM 1998 III, pages 645–656, 1998.

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Finding tours in the tsp. Preliminary chapter of a planned monograph on the TSP, available at URL: `http://www.caam.rice.edu/~keck/reports/lk_report.ps`, 1999.

[3] S. Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report No. CMU-CS-94-163, Carnegie Mellon University, Pittsburg, Pennsylvania, 1994.

[4] H. C. Braun. On solving traveling salesman problems by genetic algorithm. In H. P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature*, volume 496 of *Lecture Notes in Computer Science*, pages 129–133, Berlin, 1991. Springer-Verlag.

[5] E. Cantu-Paz. A summary of research on parallel genetic algoritms. Technical Report 95007, University of Illinois at Urbana-Champaign, Genetic Algoritms Lab. (IlliGAL), http://gal4.ge.uiuc.edu/illigal.home.html, July 1995.

[6] G. A. Croes. A method for solving traveling salesman problems. *Operations Research*, 6:791–812, 1958.

[7] M. Dorigo and V. Maniezzo. Parallel genetic algorithms: Introduction and overview of current research. In *Parallel Genetic Algorithms*, pages 5–42. IOS Press, 1993.

[8] M. Gorges-Schleuter. Asparagos96 and the travelling salesman problem. In T. Bäck, editor, *Proceedings of the Fourth International Conference on Evolutionary Computation*, pages 171–174, New York, IEEE Press, 1997.

[9] J. Grefenstette, R. Gopal, B. Rosimaita, and D. van Gucht. Genetic algorithms for the traveling salesman problem. In *In* Proceedings of an International Conference on Genetics Algorithms and their Applications, pages 160–168, 1985.

[10] G. Harik, , D. Goldberg, and B. Miller. The gamblers ruin problem, genetic algorithms, and the sizing of populations. In T. Bäck, editor, *Proceedings of the Fourth International Conference on Evolutionary Computation*, pages 7–12, New York, IEEE Press, 1997.

[11] G. Harik, F. Lobo, and D. Goldberg. The compact genetic algorithm. Technical Report No. 97006, University of Illinois at Urbana-Champaign, Urbana, IL, 1997.

[12] J. H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, Ann Arbor, MI, 1975.

[13] D. S. Johnson and L. A. McGeoch. *Local Search in Combinatorial Optimization*, chapter The Traveling Salesman Problem: A Case Study in Local Optimization. John Wiley and Sons, New York, 1996.

[14] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[15] S. Lin. Computer solution of the traveling salesman problem. *Bell Syst. Tech. J.*, 44:2245–2269, 1965.

[16] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

[17] O. Martin and S.W.Otto. Combining simulated annealing with local search heuristic. To appear on Annals of Operation Research.

[18] O. Martin, S.W.Otto, and E.W. Felten. Large step markov chain for the traveling salesman. *J.Complex Syst. 5:3:299*, 1991.

[19] P. Merz and B. Freisleben. Genetic local search for the TSP: New results. In *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*, pages 159–163, Indianapolis, USA, 1997. IEEE press.

[20] Z. Michalewicz. *Genetic Algorithms + data structures = Evolution Programs*. Springer-Verlag, 1994.

[21] M. Mitchell. *An introduction to Genetic Algorithms*. The MIT Press, Cambridge, Massachusetts, 1996.

[22] M. Padberg and G. Rinaldi. Optimization of a 532-city symmetric genetic traveling salesman problem by branch & cut. In *Operations Research Lett. 6*, pages 1–7, 1987.

[23] G. Reinelt. TSPLIB—A traveling salesman problem library. *ORSA Journal on Computing*, 3:376–384, 1991.

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

[25] G. Syswerda. Simulated crossover in genetic algorithms. In L. D. Whitley, editor, *Foundation of Genetic Algorithms 2*, pages 239–255, San Mateo, CA, 1993. Morgan Kaufmann.

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