Journal of Heuristics, 10: 483–506, 2004

A New Memetic Algorithm for the Asymmetric Traveling Salesman Problem

LUCIANA BURIOL
PAULO M. FRANÇA*
Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, UNICAMP, 13083-970 Campinas, SP, Brazil
email: buriol@densis.fee.unicamp.br
email: franca@densis.fee.unicamp.br

PABLO MOSCATO
School of Electrical Engineering and Computer Science, Faculty of Engineering and Built Environment, The University of Newcastle, 2308 Callaghan, NSW, Australia
email: moscato@cs.newcastle.edu.au

Submitted in October 1999 and accepted by Anthony Cox, Jr. in April 2002 after 2 revisions

Abstract

This paper introduces a new memetic algorithm specialized for the asymmetric instances of the traveling salesman problem (ATSP). The method incorporates a new local search engine and many other features that contribute to its effectiveness, such as: (i) the topological organization of the population as a complete ternary tree with thirteen nodes; (ii) the hierarchical organization of the population in overlapping clusters leading to the special selection scheme; (iii) efficient data structures. Computational experiments are conducted on all ATSP instances available in the TSPLIB, and on a set of larger asymmetric instances with known optimal solutions. The comparisons show that the results obtained by our method compare favorably with those obtained by several other algorithms recently proposed for the ATSP.

Key Words: asymmetric traveling salesman problem, local search, memetic algorithms, metaheuristics

1. Introduction

The Traveling Salesman Problem (TSP) is the problem of finding the shortest closed route among n cities, having as input the complete distance matrix among all cities. Let c_{ij} be a non-negative integer that stands for the *cost* to travel from city i directly to city j. A *symmetric* TSP (STSP) instance is any instance of TSP such that $c_{ij} = c_{ji}$ for all cities i, j. An *asymmetric* TSP (ATSP) instance is any instance of TSP that has at least one pair of cities such that $c_{ij} \neq c_{ji}$. The ATSP can formally be stated as follows: given as input a complete directed graph $G = (V, A)$, where $V := \{1, \ldots, n\}$ and $A := \{(i, j) : i, j \in V, i \neq j\}$ are the set of vertices and arcs of G, respectively. A feasible solution for the ATSP is a

*Author to whom all correspondence should be addressed.

Hamiltonian circuit (*tour*). The objective is to find a tour of the minimum total length, where the length is the sum of the costs of each arc in the tour. This said, the ATSP is just the special case of the problem on which we restrict the input to asymmetric instances.

The TSP has attracted a great deal of attention among researchers in recent decades. In fact, it has been used as one of the most important test-beds for new combinatorial optimization methods. Its growing popularity is also due to several important real world applications, mainly in shop floor control (scheduling), distribution of goods and services (vehicle routing), product design (VLSI layout), etc. Exact algorithms have been proposed for both symmetric and asymmetric cases. Since the TSP has proved to belong to the class of *NP*-hard problems (Garey and Johnson, 1979), heuristics and metaheuristics occupy an important place in the methods so far developed to provide practical solutions for large instances. For surveys on solution methods for the TSP, the reader may refer to Balas and Toth (1985), Laporte (1992), and Jünger, Reinelt, and Rinaldi (1995). We also strongly recommend Gutin and Punnen (2002).

Focusing only on exact methods for the ATSP, the *branch-and-bound* algorithm proposed by Miller and Pekny (1991) uses the well known *Assignment Problem* (AP) relaxation of the ATSP. Optimal solutions are reported for instances of up to 500,000 cities in reasonable running times, although such instances were randomly generated from a uniform distribution on a given interval. In such a case, the instances seem to be easy for AP-based algorithms (Fischetti and Toth, 1997). Using a similar approach, Carpaneto, Dell'Amico, and Toth (1995) solved problems with up to 2000 nodes in less than 1 minute. However, there are instances in which AP-based algorithms may face problems, such as those where costs are almost symmetric (i.e. $c_{ij} \approx c_{ji}$ for all i, j). Another class of difficult instances comes from real-world situations related to vehicle routing problems, such as those which arise in pharmaceutical product-delivery tasks within the city of Bologna (Fischetti, Toth, and Vigo, 1994). These ATSP instances and their optimal tour costs are available in the TSPLIB (Reinelt, 1991) and have been used in this paper. The polyhedral approach used by Fischetti and Toth (1997) performs better for these two classes of hard instances than AP-based methods. In the computational experiments related here, the instances of these two classes were used to evaluate the proposed method.

On the heuristic side, a wide spectrum of metaheuristic techniques has been proposed, such as *tabu search* (Fiechter, 1994), *neural networks* (Potvin, 1993), *ant colonies* (Stützle and Dorigo, 1999) and *genetic algorithms* (GAs). Of particular interest are the GAs, due to the effectiveness achieved by this class of techniques in finding good solutions in short computational times. The best results have been obtained by the combination of evolutionary algorithms with individual search methods. This hybrid population-based approach, also known as *memetic algorithms* (MAs) (Moscato, 1989, 1999; Moscato and Norman, 1992), combines the recognized strength of population methods with the intensification capability of a local search. In an MA, all agents evolve solutions until they become local minima of a certain neighborhood (or highly evolved solutions of individual search strategies), i.e., after steps of recombination and mutation, a local search is applied to the resulting solutions.

Several MAs for solving the ATSP have recently been proposed. Based on the computational results reported for these methods, one can say that a good MA implementation must combine several essential features: (i) suitable recombination and mutation operators; (ii)

a fast and effective local search algorithm; (iii) a hierarchically structured population; (iv) suitable data structures and smart codification mechanisms. The first feature is obviously inherent in any genetic algorithm, while the second is crucial since most of the CPU time is generally spent in the local search procedure. Many experiments conducted in previous research have shown that the adoption of structures in which agents are constrained to recombine within hierarchically structured subpopulations have proved to be more effective than non-structured implementations (Moscato, 1993; Gorges-Schleuter, 1997; França, Mendes, and Moscato, 2001; Berretta and Moscato, 1999).

Permitting unfeasible solutions into the population is not common in evolutive algorithms, but it was a strategy adopted by Choi, Kim, and Kim (2003) in their genetic algorithm. The population is initially generated without considering the sub-tour constraint and is composed of 200 individuals. The population is sorted by fitness values and structured in three classes where in the first class are the 40% best and in the last class the 20% worst fitness individuals. Two crossover operators are used: partially matched crossover (PMX) and tie break crossover (TBX). A *3-opt* move is applied as the local search procedure. The Karp patching algorithm is applied to repair unfeasible solutions.

The *GLS* (Genetic Local Search) method proposed by Freisleben and Merz (1996) is a successful MA that introduces a new recombination operator, called Distance Preserving Crossover (DPX). Their algorithm uses the well-known *Lin-Kernighan* (LK) heuristic as a local search engine for the Euclidean (symmetric) instances and the 3-*opt* variant for the asymmetric ones. In a more recent article, they improved the results by adopting a series of sophisticated implementation mechanisms that enhanced the performance of the method (Merz and Freisleben, 1997) as, for instance, by adding a variant of *4-opt* move to the search in the ATSP case. The mutation operator reverses a randomly chosen sub-path of length 6 (performing a random 7-change on the current tour). The mutation and crossover rate is set to 0.5, i.e. 40 individuals are created per iteration.

Gorges-Schleuter (1997) has chosen to invest in spatially structured population. In this algorithm, called *Asparagos96*, the population is organized in *demes* (local populations) spatially disposed as a ring and receiving the name of *ladder-population* (Gorges-Schleuter, 1989). The local search algorithm employed is a *3-opt* variant. The recombination operator used is MPX2, which differs slightly from the previously reported Maximal Preservative Crossover (MPX). The construction heuristic differs from the original by allocating a probability of 2/3 for the choice of the nearest neighbor and 1/3 for the choice of the second nearest neighbor. The mutation performs a non-sequential *4-opt* move. Limited computational tests carried out with these two approaches on a few instances of the TSPLIB showed that for larger instances of the STSP, *Asparagos96* is superior, although the *GLS* performs better in the case of up to 783 cities. For the ATSP, *Asparagos96* outperformed *GLS* in all 5 instances used in the comparison. More recently, Merz and Freisleben (2001) present outstanding results using a MA designed for the Euclidean instances of the TSP.

In 1997, Nagata and Kobayashi proposed a recombination operator called EAX (Edge Assembly Crossover) (Nagata and Kobayashi, 1997). This operator is appropriate for use in both the STSP and the ASTP and able to generate a wide variety of individuals from a single pair of parents. For each pair of parents, children are generated by the EAX until an improved individual is found within the limit of 100 trials. Heuristic information is added during the

```
begin
    initializePop();
    repeat /* generation loop */
        structurePop();
        recombinePop();
        mutatePop();
        optimizePop ();
    until TerminationCondition = satisfied;
end
```

Figure 1. MA structure proposed.

recombination process in such a way that the local search procedure can be disregarded. The algorithm, however, can be considered to be an MA, since it aggregates additional knowledge during the evolution process. The initial population is randomly generated.

Another algorithm employing the same "big family" concept, and based on the *soft brood selection* method used in genetic programming, was developed by Walters (1998). In brood selection, two parents may generate many individuals but only the best with respect to their relative fitness values are selected for the rest of the procedure. In fact, one can choose more than one offspring per iteration. The well known two-point recombination operator and the chromosome repair Directed Edge Repair (DER) are combined with the *3-opt* variant and successfully applied to solve a very reduced subset of instances of the ATSP drawn from the TSPLIB. The recombination operator used by Walters (1998) creates 20 individuals but only 0, 1 or 2 of them will be considered for the next generation. The local search is a *3-opt* variant and the mutation is applied with 20% probability; each mutation consists of a change of the values from 4 to 7 genes using a neighbor index distribution.

In this paper we propose a new MA for solving the ATSP that incorporates all of the four key features responsible for an effective solution. Figure 1 presents a general structure of the MA proposed in this paper.

In each generation of our proposed MA, the population is re-structured by *UpdatePocket* and *PocketPropagation* procedures, represented by *structurePop()* procedure in figure 1. The population is hierarchically organized as a complete ternary tree of 13 agents clustered in 4 subpopulations, which in turn are composed of four individuals. Thirteen new individuals are generated for the recombination operator loop by *recombinePop()* procedure. Each new individual has a mutation probability of 5%, executed by m*utatePop()* procedure. In the end, the local search, which requires an ordered list of the 5 nearest neighbors of each city, is applied for each of these new individuals by *optimizePop()* procedure. The main contribution is a new local search, called Recursive Arc Insertion (RAI), especially designed for the asymmetric case. A new recombination operator, Strategic Arc Crossover (SAX), adapted from a similar recombination operator called Strategic Edge Crossover (SEX) (Moscato and Norman, 1992), was found to be capable of dealing with asymmetric instances. The data structures and algorithms used allow near-optimal solutions to be found rapidly.

The present paper is organized as follows. Section 1 presents an overview on the problem while in Section 2, the data structures used to code the proposed MA are presented. In Section

3 the operators used by the MA are presented, such as the new local search algorithm and recombination operators. Section 4 introduces the procedures used to avoid fast convergence of the population and the paper finishes with Section 5, where the proposed method is evaluated by a series of computational experiments using two sets of data drawn from the literature. Empirical comparisons with seven other leading heuristics for this problem are also reported.

2. Data structures

First, the data structures used by the proposed MA will be described. They are considered separately since they contain several features highly reusable in other problem domains.

2.1. Solution representation

The data structure adopted to represent an ATSP tour consists of a bidimensional array with 2 rows and n columns. Each index i of the array corresponds to the city i, and the array keeps the indices associated with the predecessor (*Prev)* and the successor (*Next*) cities of this city i. The reasons for choosing this data structure are two-fold. First, due to its undeniable simplicity and secondly, the fact that numerous *3-Arc* changes recursively performed by the local search RAI can be executed in constant time.

2.2. Agent concept

The population of our MA is composed of a set of entities called *agents*. Each agent handles two feasible solutions, which will be referred to as the *Pocket* and *Current* solutions (*Pocket-Current* methodology has also been used in other work, e.g., Moscato and Tinetti, 1992; Paechter et al., 1996; Berretta and Moscato, 1999). In the MA proposed here, in each generation the recombination phase uses *Pocket* and *Current* solutions as parent and 13 new individuals are created, which substitute the *Current* solutions in the population.

2.3. Population structure

The population structure of this method (figure 2) is composed of 13 agents, which are organized as a ternary tree of three levels and can be understood as a variant of *island* models. However, since the working metaphor of MAs is related to cultural evolution, we can also think of it as a hierarchical peer-reviewed organization. In this implementation, there is no *migration mechanism* of agents between subpopulations (where we think of each subpopulation as consisting of four agents). Agents at an intermediate level (those which are neither part of the root nor the leaves of the tree) are concomitantly part of two different subpopulations. Structured populations have been also used by Berretta and Moscato (1999).

Each subpopulation is composed of a single *leader* and three *supporter* agents, the latter located one level below in the hierarchy. Agent 1, located at the root of the tree, is the leader

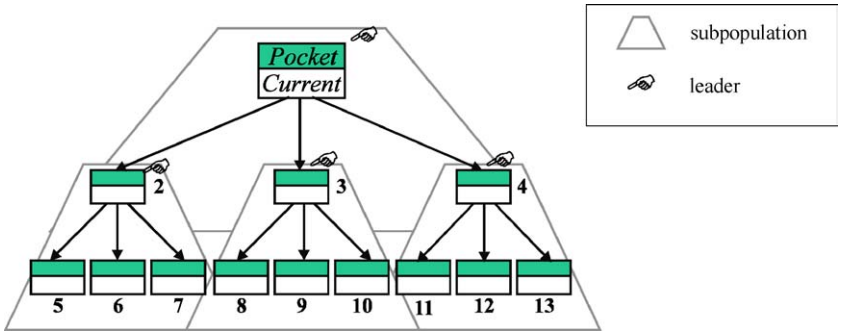

Figure 2. Population structure.

of the top subpopulation and is supported by Agents 2, 3, and 4. Agent 2 is supported by Agents 5, 6, and 7, etc. We remark the fact that agents 2, 3, and 4 (being in an intermediate level) play both leader and supporter roles.

Before the recombination phase, the tree is re-structured using two operations: *UpdatePocket* and *PocketPropagation*. When the cost of the *Current* solution is better than that of a pocket solution from the same agent ($curCost < pocCost$) and the population does not have a *Pocket* solution with cost equal to *curCost*, the solutions are switched. Thus, it plays a role of a memory of good solutions. After the *UpdatePocket*, the *PocketPropagation* procedure is applied. This procedure exchanges the *Pocket* solutions of a leader and one of its supporters, if the latter has a *Pocket* solution which is better than that of the leader. Note that these two mechanisms are all that is needed to guarantee the flow of better solutions towards the agent at the top of the hierarchy (root node). Both mechanisms are inherently asynchronous, which accounts for the suitability of the method for distributed processing implementation.

3. Operators of the memetic algorithm

In this section, the operators for the specific problem to be solved, namely the ATSP, are introduced. However, this specific use does not limit the application of these procedures to other problem domains. For instance, the recombination operators or the local search engines can be applied to other problems suitable for permutation-based representations, such as scheduling problems (Mendes et al., 2001).

3.1. Initial population

There are many possible choices for the initial population. One of them is to create it entirely at random, while another possibility would be to apply some greedy constructive heuristic. Zhang (1993, 2000) has proposed to solve the Assignment Problem (AP) and to apply the Karp's patching (1979) to construct a feasible tour. The AP is a relaxation of the ATSP, since

the assignments need not form a complete tour. Zhang has concluded that, in general, the quality of the tours from the AP + patching algorithm is more than one order of magnitude smaller than those from the other methods, on all problem structures he had tested.

We have chosen to construct the *Pocket* solution from Node 1 (root node in figure 2) using the AP + patching heuristic proposed by Zhang, while all other solutions are generated using the nearest neighbor heuristic. Since the last cities inserted are generally linked to distant neighbors, it is interesting to use these distant neighbors as new starting points for the construction of new initial solutions. The aim is to generate a diversified initial population. Thus, for each subpopulation, the three last cities inserted in the nearest neighbor solution for the corresponding *Pocket* and *Current* of the leader are considered to be the starting cities for the construction of the *Pocket* and *Current* solutions of the corresponding supporters.

3.2. Recombination operators

The fact that this article proposes a new local search procedure for the ATSP made it necessary to test its quality *vis-à-vis* various recombination operators. It is well known that the performance of an MA is dependent on a good synergy between the local search algorithm and the recombination operator utilized. From the vast literature on recombination operator proposals, four TSP-based recombination operators were selected: (i) the DPX recombination operator of Freisleben and Merz (1996); (ii) the MFNN, introduced by Holstein and Moscato (1999); (iii) the Uniform Nearest Neighbor Crossover (UNN) operator, and (iv) the SAX operator, a generalization of SEX that was originally proposed for the STSP (Moscato and Norman, 1992). In the following section, the basic idea behind the derivation of each one of the operators will be presented, as well as the adaptation leading to the SAX operator.

3.2.1. Distance preserving crossover (DPX). The functionality of the DPX (Freisleben and Merz, 1996) stems from the observation that the average distance between locally optimal tours is closely related to the average distance between a locally optimal tour and the global optimum. In this case, the distance between two TSP tours represents the number of arcs belonging to one tour but not to the other. The aim of the DPX operator is to construct an offspring that has the same distance to each of its parents as the parents have to each other. This can be accomplished by copying the common genes of the two parents onto the offspring. The tour fragments are then reconnected using the nearest neighbor heuristic, but *without* using any of the arcs contained *in only one* of the parents. More formally, let $A \cap B$ be the set of arcs that parents A and B have in common, and $A - B$ the set of arcs that belong to A but not to B. Hence, the set of arcs used to generate the offspring is $A \cap B + E - (A - B) \cup (B - A)$, where E represents the complete arc set.

3.2.2. Multiple fragment-nearest neighbor repair edge recombination (MFNN). This recombination operator (Holstein and Moscato, 1999) employs a different strategy for the construction of offspring. Here the set of arcs used to build the offspring are all the arcs in $A \cap B$ (as in the DPX). In one single step, they are all added to the new solution. Afterwards, those arcs belonging to $(A - B) \cup (B - A)$ are added (whenever possible) in non-decreasing

order of priority. This means that after copying the common arcs from the parents A and B onto the offspring, the remaining arcs are selected from those that are contained in at least one of the parents. Arcs are inserted following a non-decreasing order in relation to cost, taking into account the non-violation of assignment and sub-tour elimination constraints. The nearest neighbor algorithm is then applied to reconnect the fragments and construct a valid tour.

3.2.3. Uniform nearest neighbor crossover (UNN). The Uniform Nearest Neighbor Crossover is an adaptation of the uniform crossover operator (Goldberg, 1989); in this original procedure an offspring is generated by a random choice of genes from the two parents. Each gene value depends on the outcome of a Boolean variable: if *true*, the corresponding gene of parent A is copied onto the offspring, otherwise, the copy comes from parent B. In the presented UNN implementation, initially all arcs in common between both parents are copied onto the offspring. The remaining Arcs are inserted as follows: if the outcome corresponding to the city i is *true*, then the arc which links i to the next city in parent A is copied to the offspring, since it does not violate any restriction. If it is *false*, then the arc to be copied belongs to parent B. In either case, if a violation occurs, then the arc of the other parent is tested. The resulting tour fragments are patched using the nearest neighbor heuristic.

3.2.4. Strategic arc crossover (SAX). As stated before, SAX is an adaptation of SEX, originally proposed for the STSP. Historically, SEX can be considered to be a direct descendant of the *Enhanced Edge Recombination* procedure proposed by Whitley (1991) in combination with a heuristic introduced by Karp (1977, 1979). In the heuristic, the process that creates a new offspring can be better understood by describing it in two phases: generation of strings (a string is a path with η cities, $1 \leq \eta \leq$ N), called *CreateStrings*, and application of the *PatchStrings* procedure (the strings are patched to form a valid ATSP tour).

Given tours A and B, a graph G is constructed, consisting of the union of the arcs in the two tours, but avoiding repeating arcs. Thus, each vertex in the resulting graph will have out-degree 1 or 2 and in-degree 1 or 2, with strings generated by *CreateStrings*, as shown in figure 3.

The algorithm used to recover a valid ATSP tour after the procedure *PatchStrings* is the nearest neighbor tour construction heuristic. Note that this recombination operator does not guarantee that a common arc from both parents is inherited by the respective offspring. For example, if both parents have in common the arcs *a-b-c* and the initial vertex selected in Step 1 is *c*, the result can be the string *c-x-a-b* (with *x* being a string). Then the arc *b-c* cannot be added to this string, although it can be added by the *PatchStrings* procedure.

Among the recombination operators presented here, certainly UNN and SAX are the most similar. Both operators maximize the offspring arcs inherited from the parents, and neither uses any specific ordering as the MFNN. But there are two important differences between SAX and UNN. First, SAX generates one string at a time, starting from a randomly selected unvisited city, and it attempts to increase the string using arcs from the parents, whereas UNN generates all strings simultaneously. Second, in contrast to UNN, SAX does not guarantee that the offspring will inherit all the common arcs from the parents. Another

Step 1: Pick a random unused vertex v in G as the initial vertex for a new string, mark it as used, and declare it to be the current string's tail t and head h.

Step 2: While the head h of the current string has at least one out-arc to an unused vertex, do the following: randomly choose one of the unused vertices to which h has an out-arc, add it to the end of the list, mark it as used, and declare it to be the new head of the string.

Step 3: Declare the current string head h to be the end of the string.

Step 4: While the tail t of the current string has at least one in-arc from an unused vertex, do the following: randomly choose one of the unused vertices to which t has an in-arc, add it to the beginning of the list, mark it as used, and declare it to be the new tail of the string.

Step 5: Declare the current string tail t to be the beginning of the string. If there is at least an unused vertex, go to Step 1.

Figure 3. *CreateStrings* procedure from SAX.

UNN operator without the procedure that copy all arcs in common between both parents onto the offspring was tested, but the quality of the results did not change while the time increased.

Empirical results showed that, on average, the SAX operator outperformed the UNN in our experiments. It can thus be concluded that the SAX operator is more effective.

3.3. The new local search algorithm: Recursive Arc Insertion (RAI)

Several neighborhood definitions are used in local search-based approaches for the TSP, although the literature generally refers to the Lin-Kernighan neighborhood as the most powerful single-agent local search optimizer (Lin and Kernighan, 1973). Several variations of this approach have shown themselves to be useful, especially in large geometric instances. For asymmetric instances, the heuristic proposed by Kanellakis and Papadimitriou (1980), (and some variants of it) has proved to be useful, despite its worst-case polynomial time bound of something like $n^5 \log(n)$. However, most researchers have preferred to use simpler neighborhoods, like the *k-opt* neighborhood reduced, due to its good performance. For instance, Johnson and McGeoch (1997) presented a *2-opt* and a *3-opt* variants, based on a clever neighborhood reduction and with good results for the symmetric case. In the present paper, a new local search procedure called *Recursive Arc Insertion* (RAI) is used, with basic transitions always contained in the 3-o*pt* neighborhood.

The rationale underlying this new local search heuristic is to improve the tours starting from cities marked with *don't look bits* (Bentley, 1990) during the recombination and mutation phases. In the recombination step, all cities in the extremes of the strings have their *don't look bits* assigned as *false*. In the mutation step, all the cities at the extremes of the arcs involved in the change (an insertion move) will also have their *don't look bits* set to *false*.

The cities with *don't look bits* set to *false* are considered to be "critical" cities and are the *starting points* for triggering the search. Before the recombination operator, and after the

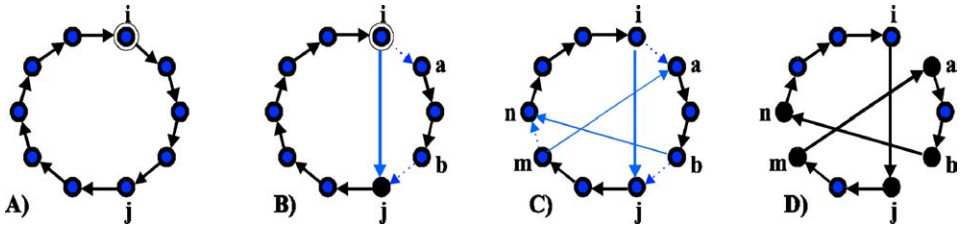

Figure 4. Illustration of a RAI move.

local search, all the cities have the *don't look bits* set to *true*. In a given tour, a single pass is made through each starting point (though due to the recursive nature of the algorithm, the search may consider any specific city more than once).

Given a city i, we define the s-out-neighbor(i) as the set of the destination cities of the $s$shortest outgoing arcs from the city i. Analogously, the p-in-neighbor(i) is constructed by the set of the cities of the p shortest incoming arcs to the city i. Let i be a critical starting city selected in a given tour. Figure 4 helps to understand the algorithm steps.

Step 1. (Figure 4A) Choose $j \neq Next(j)$, $\in$ p-in-neighbor(i). Set $Count = 0$ and go to Step 2.

Step 2. (Figure 4B) Consider a trial move adding arc (i, j) and deleting arcs (i, a) and (b, j), where $a = Next(i)$ and $b = Prev(j)$. Calculate $\Delta_1 = c_{ia} + c_{bj} - c_{ij}$. Set $Count = Count + 1$ and go to Step 3.

Step 3. (Figure 4C) Starting from arc $(j, Next(j))$, select the first arc $(m, n) \neq (i, j)$ in the subtour $\{j, \ldots, i, j\}$ such that $\Delta_2 < \Delta_1$, $\Delta_2 = c_{ma} + c_{bn} - c_{mn}$. If it finds such an arc (m, n), go to Step 4. If $\Delta_2 \geq \Delta_1$ for all arcs in the sub-tour $\{j, \ldots, i, j\}$, go to Step 5.

Step 4. (Figure 4D) Perform the move, e.g., insert arcs (i, j), (m, a) and (b, n) and remove arcs (i, a), (b, j) and (m, n). Invoke the procedure recursively using as initial critical city (city i) the cities a, b, n, m, j and i, in that order. Go to Step 5.

Step 5. If $Count = 2$, stop. Otherwise, select $j \neq Next(i) \in s$-out-neighbor(i). Set $Count = 2$ and exchange the indices i and j (by doing $x = i, i = j$ and $j = x$), and go to Step 2.

Both stack and recursion disciplines could be used, but recursion is quicker because the recursive call is very simple, with only three variables being re-allocated. Both lists and stacks were tested, but although the average results were similar, specific instances led to different results and, thus, recursion was considered to be a better alternative to be used.

In this local search, the s-out-nearest neighbors and the p-in-nearest neighbors of each city i are stored in a two-dimensional matrix in a non-decreasing order of their distances in order to facilitate the search for city j in Steps 1 and 5. Computational experiments conducted in Section 5 led to the best choices for parameters p and s. In Step 1, the closer the neighbor j is, the greater is the probability to be selected. The best probability values are also set in Section 5. Note that during the execution of Step 2, the insertion of arc (i, j) and the deletion of arcs (i, a) and (b, j) have created a subtour and a string of arcs. Note, however, that the string formed may have several, one, or even no arcs (in the case $a = b$).

Following, the first arc (m, n) in the subtour that causes a tour improvement is detected and the feasibility is reestablished. Obviously, $(m, n) \neq (i, j)$, since we are testing for insertion of the new arc (i, j). The alternative strategy, looking for an arc (m, n) with the best improvement, was tested but discarded since the results showed a lack of diversity in the population. Tests considering a fixed number of arcs also proved to be unsuccessful. We believe that what most contributes to the strength of the search is an adequate combination of a greedy choice of arcs involved in $\Delta_1$calculation, with an extensive search for arcs relative to Δ_2 calculation.

4. Maintaining population diversity

Robustness is a key element in the development of any metaheuristic. Hence, a method that shows an effective behavior, namely, attaining very near optimal solutions in short CPU time regardless of the instance size (and the particular type of instance) being solved, is highly desirable. Of equal relevance is the ability of the method to solve a set of instances with the same parameter settings previously used in the solution of other instances, with a large step towards this goal probably being the adoption of a population structured as a ternary tree. In all the experiments carried out with the present MA, the population size was set at 13 agents (26 individuals). Compared with other MAs already proposed for the TSP, this population is remarkably small, but this makes it possible to perform a much larger number of generations in the same time span.

To maintain the population diversity, the MA implementation makes use of some specific procedures concerning selection for mutation and recombination.

In the context of an MA, mutation is used for the purpose of maintaining population diversity. In each generation, any specific *Current* individual has a 5% probability of being selected. If it is, a *one-city insertion* procedure is applied to the tour considering random cities. The five cities involved in the procedure have their *don't look bits* marked for the execution of the local search.

The selection for recombination procedure requires a suitable choice of agents to preserve population diversity. Consider the additional notation used for each subpopulation of complete tree of degree k. For example, $k = 3$ for the ternary tree from figure 2, $k = 2$ for a binary tree, etc. Each subpopulation is composed by $k + 1$ nodes: one in a higher hierarchy and k in a lowest hierarchy.

- *leader*: the higher in the hierarchy of a given subpopulation;
- $offspring_1$, $offspring_2$, ..., $offspring_k$: the supporter agents of a subpopulation;
- *Pocket*(i): the *Pocket* solution of agent i;
- *Current*(i): the *Current* solution of agent i.

Remembering that each recombination replaces a previous *Current* solution, the recombination operations for each subpopulation follow:

- *Current* of the *leader* $\Leftarrow$ *Recombine* (*Pocket* (*leader*), *Current* ($offspring_1$));
- *Current* of $offspring_k$ $\Leftarrow$ *Recombine* (*Pocket* ($offspring_k$), *Current* (*leader*));

- *Current* of $offspring_{i=1,2,\ldots,(k-1)} \Leftarrow$ *Recombine* (*Pocket* ($offspring_i$), *Current* ($offspring_{(i+1)}$));

The choice of the suporters $offspring_i$ is made at random.

Another diversity feature is to admit in the population only *Pocket* solutions with different fitness values. To maintain this characteristic, the procedure *UpdatePocket* is only applied in an agent i in case its *curCost* is different from the *pocCost* of any *Pocket* solution of the population.

5. Computational experiments and results

This section presents the different computational experiments performed using the proposed MA. The code was written in the Java programming language using the *Java Development Kit 1.3* on a laptop Pentium 1.7 GHz running GNU/linux. The algorithm stops in generation $\lfloor 13 * \log(13) * \log(n^2) \rfloor$ or if it has run for 100 generations without improving the incumbent solution. Furthermore, if the construction heuristic AP + patching generates only one subcycle, it indicates that the ATSP problem was solved to optimality, and the algorithm stops. All results obtained with our MA and presented in this paper are an average of 20 runs using the same algorithm and parameters. The computational time reported includes the entire procedure shown in figure 1.

As mentioned before, the purpose of the first computational experiment was to test the performance of the MA when RAI local search acts in conjunction with 4 different recombination operators. The instance set used in the computational experiments is composed by the 27 ATSP instances available in the TSPLIB (Reinelt, 1991). Results attained for each recombination operator are shown in Table 1, where *Opt/Trials* represents the average number of times the method succeeded in finding the optimal solution over 20 trials, *Gap(%)* means the average percentage deviation (quality) from optimal solutions ($100 * ($*Heuristic -Optimal)/Optimal*), *Gen* is the average number of generations performed and *CPU Time* denotes the average CPU time in seconds.

For better visualization, the most important criteria, namely *Gap(%)* and *CPU Time*, are depicted in figure 5. The best tradeoff was attained by the SAX recombination operator, which was correlated with the number of times optimal solutions were found.

In the second computational experiment the best values for parameters were set. We selected three main parameters to conduct the study, namely, the population size, the type

Table 1. Average MA performance using RAI with 4 different recombination operators.

Recombination operator	Opt/Trials	Gap(%)	Gen	CPU time
SAX	16.70	0.04	112.88	1.07
DPX	15.93	0.06	111.34	1.97
MFNN	5.48	0.75	115.97	0.45
UNN	10.04	0.21	114.42	0.62

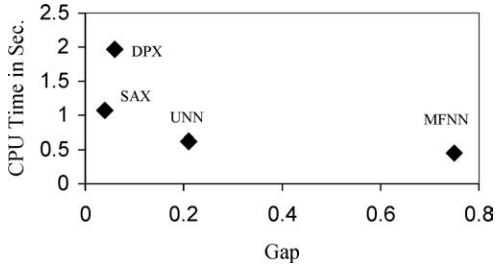

Figure 5. MA performance with respect to *Gap(%)* and *CPU time*.

of tree structure and the neighborhood size used by the local search procedure. Remembering that the population is always conceived as a complete tree, there are two variables involved in the tree structure definition: the degree of each node and the number of levels of the tree. For instance, if the node degree is set to 2 (binary tree), a 2-level tree will have 7 agents while a 3-level tree will have 15 agents.

For the neighborhood list size, which is controlled by parameters p and s, we fixed the values $p, s = 4, 5, 6$ and 10 to be tested. The probabilities of selection of the nearest neighbor, the second nearest, and so on are fixed as shown in Table 2.

Table 3 summarizes the results. *Gap(%)* represents the average percentage derivation from optimal solutions over 20 runs for each one of the TSPLIB instances. *CPU Time* is the average time in seconds. *Pop Size* means the total number of agents that compose the populations. D*egree/Level* denotes the node degree and the number of levels of the tree, and p, s is the neighborhood list size. Table 3 is arranged in increasing order of *Gap*.

For better visualization the values from the non-dominated combination with $Gap \leq 0.25$ are depicted in figure 6. We consider that a combination i is non-dominated if $Gap_i < Gap_j$ and/or $CPUTime_i < CPUTime_j$ for any other combination j from Table 3. We concluded that a suitable combination of parameters is *Pop Size* = 13, *Degree/Level* = 3/3 and $p, s = 5$ (pointed in figure 6), although some other parameter combinations have comparable tradeoffs.

We emphasize that the parameter combination chosen above is the only one used by our implementation in all other subsequent computational tests and comparisons.

Table 2. Probability of selection of the first $p = s$ nearest neighbors used by the local search.

p, s	Probabilities (%)
4	40, 30, 20, 10
5	40, 30, 15, 10, 5
6	35, 20, 18, 13, 9, 5
10	20, 18, 15, 13, 10, 8, 6, 5, 3, 2

Table 3. Parameter setting results.

Gap (%)	CPU time	Pop size	Degree/Level	p, s
0.003	12.24	85	4/4	6
0.004	11.75	85	4/4	4
0.006	17.14	85	4/4	5
0.006	5.43	43	6/3	6
0.009	7.64	43	6/3	5
0.012	19.03	85	4/4	10
0.012	4.72	43	6/3	4
0.012	5.53	40	3/4	4
0.012	5.91	31	5/3	10
0.012	7.70	43	6/3	10
0.014	3.89	31	2/5	4
0.015	5.94	40	3/4	5
0.016	3.45	31	5/3	4
0.016	4.22	31	5/3	5
0.017	9.05	40	3/4	10
0.018	3.67	31	5/3	6
0.019	5.02	40	3/4	6
0.021	4.53	31	2/5	5
0.021	4.62	31	2/5	6
0.023	2.43	21	4/3	6
0.026	2.86	21	4/3	4
0.026	4.45	21	4/3	10
0.028	7.33	31	2/5	10
0.029	2.2	21	4/3	5
0.032	1.95	15	2/4	6
0.035	1.31	13	3/3	4
0.037	1.62	15	2/4	4
0.039	1.94	15	2/4	5
0.040	**1.11**	**13**	**3/3**	**5**
0.041	1.3	13	3/3	6
0.044	1.98	15	2/4	10
0.068	1.56	13	3/3	10
0.196	0.32	7	2/3	5
0.215	0.24	7	2/3	4
0.226	0.26	7	2/3	6
0.289	0.28	7	2/3	10
1.852	0.1	7	6/2	5
2.005	0.07	7	6/2	6
2.038	0.08	7	6/2	4
2.145	0.13	7	6/2	10

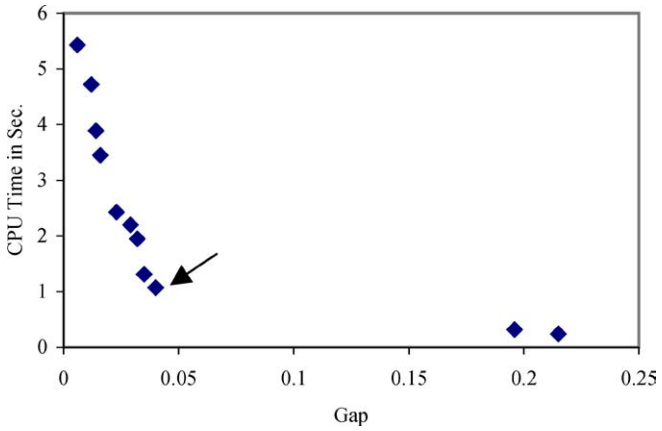

Figure 6. Time-quality tradeoff for non-dominated combination with $Gap \leq 0.25$.

Table 4 reports the detailed performance of the proposed method on the complete instance set available in the TSPLIB, with 20 runs for each one of the 27 instances with sizes varying from 17 to 443 cities. *InitGap* represents the average initial quality solution of the population and *OPT Time* is the average time spent in reaching an optimal solution (if an optimal solution is not found, the final total time is considered). The average percentage deviation from the optimal solution attained by the best initial solution is 3.56%, with the local search procedure occupying 60% of the total time. For the 4 biggest instances, the construction heuristic AP + patching generates only one sub-cycle, which indicates that the optimal solution was found and, thus, the MA stops soon after having generated the initial population.

We have tested our MA considering different initial populations: (i) all initial solutions are constructed using the nearest neighbor heuristic and (ii) all initial solutions are randomly generated. For 20 runs of each instance from ATSP, we have found the same final quality solution (0.04%) achieved in Table 4, but greater CPU Time: 5.76 for (i) and 6.03 for (ii). The average initial solution is 27.34% for (i) and 406.95% for (ii). If we consider the first 23 instances (excluding the instances for which the AP + patching heuristic was able to find optimal solutions), we have CPU Time = 1.20 from Table 4, while CPU Time = 1.27 for (i) and CPU Time = 1.22 for (ii). With these results we conclude that one clear advantage of adopting the AP + patching construction heuristic is the possibility of stopping when the optimal solution was found with this heuristic. Since this heuristic solves an AP in the optimality, in case one sub-cycle it is found only, this solution is optimal for the ATSP.

The next computational experiment is a comparison of the performance of the SAX/RAI MA with seven other heuristic algorithms which have the best results found in the literature for solving the ATSP. Four of these five algorithms are MAs: those of Gorges-Schleuter (1997), Merz and Freisleben (1997), Nagata and Kobayashi (1997), and Walters (1998), while Stützle and Dorigo (1999) present a *non-standard* ant colony algorithm. With the exception of Nagata and Kobayashi (1997), Choi, Kim, and Kim (2003), Helsgaun (2000), and Zhang (2000), all the others use the *don't look bits* concept introduced by Bentley

Table 4. Computational results using SAX/RAI.

Instance	InitGap	Opt/Trials	Gap (%)	Gen	CPU time	OPT time
br17	0.00	20	0.00	94.00	0.14	0.05
ftv33	12.83	20	0.00	107.80	0.29	0.05
ftv35	0.14	20	0.00	113.30	0.33	0.08
ftv38	0.13	05	0.10	105.70	0.31	0.26
p43	0.05	11	0.01	119.65	0.48	0.35
ftv44	7.01	07	0.44	116.75	0.41	0.36
ftv47	2.70	20	0.00	110.00	0.47	0.13
ry48p	5.42	17	0.03	126.35	0.56	0.32
ft53	18.20	20	0.00	120.75	0.56	0.2
ftv55	3.61	20	0.00	117.05	0.50	0.16
ftv64	3.81	20	0.00	126.90	0.66	0.24
ft70	1.88	08	0.03	141.00	0.96	0.86
ftv70	3.33	19	0.01	137.20	0.83	0.38
ftv90	3.67	20	0.00	124.50	0.95	0.28
ftv100	3.24	20	0.00	130.65	1.20	0.40
kro124p	6.46	18	0.01	142.15	1.37	0.74
ftv110	4.7	18	0.02	148.55	1.71	0.98
ftv120	8.31	07	0.14	156.95	2.19	2.00
ftv130	3.12	18	0.01	156.45	2.18	1.30
ftv140	2.23	14	0.08	164.95	2.66	2.11
ftv150	2.3	18	0.01	161.25	2.69	1.54
ftv160	1.71	16	0.02	161.15	3.28	2.04
ftv170	1.38	15	0.05	164.75	3.92	2.78
rbg323	0.00	20	0.00	0.00	0.07	0.07
rbg358	0.00	20	0.00	0.00	0.08	0.08
rbg403	0.00	20	0.00	0.00	0.08	0.08
rbg443	0.00	20	0.00	0.00	0.09	0.09
Average	3.56	16.7	0.04	112.88	1.07	0.66

(1992), including the MA proposed in this paper. Table 5 summarizes the published results obtained by these methods on TSPLIB instances, first presenting the metaheuristics and then the heuristic ones. This table reports one decimal digit for CPU times and two decimal digits for *Gap(%)*, or less precision in case it was not originally supplied. The table lists first the six evolutionary algorithms (GAs and MAs) followed by the other three approaches, both organized in alphabetical order by the paper reference.

The first four rows identify the author, the method, the computer and the language of each algorithm, respectively, *Gap(%)* indicates the average percentage deviation from optimal solutions obtained over a certain number of runs (trials), and *CPU Time* the average

Table 5. Computational comparisons with previous methods.

Author	Choi, Kim, and Kim (2003)		Gorges-Schleuter (1997)		Merz and Freisleben (1997)		Nagata and Kobayashi (1997)		Stützle and Dorigo (1999)		Walters (1998)		Helsgaun (2000)		Johnson et al. (2001) -IKP4F-		Zhang (2000) and Johnson et al. (2002) -ZHANG1-	
Method	GA		MA		MA		MA		Ant Colonies		MA		Local search		Local search		Truncated branch-and-bound	
Computer	Pentium 550 MHz		SUN UltraSparc 170 MHz		DEC Alpha 233 MHz		Pentium 200 MHz		UltraSparc II 167 MHz		Pentium II 300 MHz		Pentium 1.7 GHz		Silicon Graphics Power Chal. 196 MHz		Silicon Graphics Power Chal. 196 MHz	
Language	C++		C		C++		C		C		C		C		C		C	
Instance	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time
br17	0.00	0											0.00	0.0	0.00	0.1	0.00	0.0
ftv33	0.00	4											0.00	0.0	3.02	0.1	3.50	0.0
ftv35	0.14	1											0.01	0.0	0.03	0.1	1.09	0.0
ftv38	0.13	2											0.08	0.0	0.00	0.1	1.05	0.0
p43	0.00	11	0.00	2.1	0.00	11							0.01	0.2	0.01	23.8	0.05	0.0
ftv44	1.30	5											0.00	0.0	1.30	0.1	0.00	0.0
ftv47	0.00	13											0.00	0.1	0.49	0.2	0.68	0.0
ry48p	0.33	5	0.00	3.5	0.20	72	0.00	3	0.00	2.3	0.00	0.4	0.00	0.0	1.92	0.2	2.17	0.0
ft53	0.00	8					0.56	5					0.00	0.1	0.01	8.0	10.96	0.0
ftv55	0.00	5											0.00	0.0	0.00	0.2	0.75	0.0
ftv64	0.00	7											0.00	0.1	0.33	0.4	0.00	0.0
ft70	0.21	20	0.00	4.4	0.00	111	0.00	10	0.00	37.2	0.00	2.0	0.00	0.1	0.02	4.8	0.47	0.0
ftv70	0.00	12											0.00	0.9	0.08	0.3	0.00	0.0
ftv90	0.00	20											0.00	0.2	0.08	0.5	0.44	0.0

(*Continued on next page.*)

Table 5. (*Continued*).

Author	Choi, Kim, and Kim (2003)		Gorges-Schleuter (1997)		Merz and Freisleben (1997)		Nagata and Kobayashi (1997)		Stützle and Dorigo (1999)		Walters (1998)		Helsgaun (2000)		Johnson et al. (2001) -IKP4F-		Zhang (2000) and Johnson et al. (2002) -ZHANG1-	
Method	GA		MA		MA		MA		Ant Colonies		MA		Local search		Local search		Truncated branch-and-bound	
Computer	Pentium 550 MHz		SUN UltraSparc 170 MHz		DEC Alpha 233 MHz		Pentium 200 MHz		UltraSparc II 167 MHz		Pentium II 300 MHz		Pentium 1.7 GHz		Silicon Graphics Power Chal. 196 MHz		Silicon Graphics Power Chal. 196 MHz	
Language	C++		C		C++		C		C		C		C		C		C	
Instance	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time	Gap (%)	CPU time
ftv100	0.00	62											0.00	0.3	0.13	0.6	0.00	0.0
kro124p	0.52	67	0.00	6.0	0.00	35	0.02	15	0.00	7.3	0.00	1.6	0.00	0.3	1.28	0.3	3.29	0.1
ftv110	0.54	61											0.00	0.4	0.12	1.0	0.00	0.0
ftv120	0.29	89											0.00	0.5	0.33	1.7	0.00	0.0
ftv130	0.59	97											0.00	0.6	0.27	1.5	0.00	0.1
ftv140	0.32	81											0.00	0.5	0.21	1.2	0.00	0.0
ftv150	0.55	88											0.00	0.5	0.40	1.2	0.00	0.0
ftv160	0.12	101											0.00	0.8	0.24	1.4	0.11	0.1
ftv170	0.22	94	0.00	9.5	0.26	100	0.00	60	0.00	56.2	0.00	7.3	0.00	0.9	0.45	1.5	0.36	0.1
rbg323	0.00	2					0.00	129					0.00	8.2	0.30	414.5	0.00	0.2
rbg358	0.00	3					0.00	172					0.00	6.7	0.79	342.3	0.00	0.2
rbg403	0.00	2					0.00	256					0.00	22.8	0.11	744.0	0.00	0.5
rbg443	0.00	4					0.00	268					0.00	17.5	0.09	908.1	0.00	0.5
Average	0.19	32.0	0.00	5.1	0.09	65.8	0.06	102	0.00	25.7	0.00	2.8	0.00	2.3	0.44	91.0	0.92	0.1
	0.04	*1.1*	*0.02*	*1.5*	*0.02*	*1.5*	*0.02*	*1.2*	*0.01*	*1.7*	*0.01*	*1.7*	*0.04*	*1.1*	*0.04*	*1.1*	*0.04*	*1.1*

execution time (in seconds) using the indicated computers, languages and stopping conditions. Depending on the specific experiment a varying number of runs were involved: Gorges-Schleuter, Merz and Freisleben and Walters executed 20 runs, while Nagata and Kobayashi did 30 runs, Stützle and Dorigo 25 runs, Choi et al. 3 runs and the IKP4F from Johnson et al. (2002) reports the average results from 5 or more runs for each instance. In the present study, 20 runs were used. The last row shows the average gap and CPU times of the various methods with those obtained by the present approach (in italic).

The ant colony (AC) algorithm proposed by Stützle and Dorigo (1999) constructs a tour moving city to city, guided by "pheromone trails" and *a priori* heuristic information. In each tour, a local search procedure based on *3-opt* moves is applied and, after that, trails are updated. Comparing results with the MAs mentioned above, the AC algorithm presents competitive results, although it requires longer computational time.

The Helsgaun heuristic (Helsgaun, 2000) is a Lin-Kernighan variant designed for symmetric instances. Thus, first the asymmetric instance is reduced into a symmetric one and then the Helsgaun heuristic is applied. Probably one of the main new features introduced by this variant is the change of the sequence of *2-opt* moves executed in the basic search step of Lin-Kernighan by a sequence of *5-opt* moves, reduced by using neighbor lists. Furthermore, the neighbor lists are computed based on distance functions modified by Lagrangian relaxation. The results reported for this heuristic presented in Table 5 were generated in our computer.

The local search method iKP4F proposed by Johnson et al. (2002) implements a variant of Kanellakis-Papadrimitriou algorithm (1980) which finds the best *4-opt* move using a dynamic programming approach suggested by Glover (1996). Furthermore, their implementation speeds the sequential search portion of the original algorithm using a neighbor list and *don't look bits*.

The truncated branch-and-bound from Zhang (1993, 2000) is an approximation method based on branch-and-bound subtour elimination, which uses the assignment problem as a lower-bound function. Recently, results using this algorithm were published in Johnson et al. (2002) with the denomination of ZHANG1.

The stopping criteria to from each one of the metaheuristics are:

- Choi et al.: 1,000 generations.
- Gorges-Schleuter: 80 generations.
- Merz e Freisleben: number of generations depending on the instance.
- Nagata e Kobayashi: not described (unclear to us).
- Stützle e Dorigo: maximal running time depending on the instance. The times reported are the average times to find the best solutions.
- Walters: maximal number of generations set to 500. Since the reported average generation number is smaller than 500 when the optimal solution is found, we inferred that the algorithm stopped when a optimal solution is found.

Before analyzing the results, it must be noted that the tests were made using different computers and languages/compilers. The performance of different computers is hard to compare, however, the Java compiler used in the experiments presented in this paper should optimistically be estimated to be at least 25% slower than most of the GNU C/C^{++} compilers.

Nevertheless, the average results in Table 5 suggest a superiority of the presented MA approach when compared with other metaheuristics, even though for most of them the comparisons have been made for only a limited number of instances. One can say that the methods of Gorges-Schleuter and Walters are comparable in terms of solution quality and execution time, but the TSPLIB instances used in their experiments are considered to be relatively easy. Note that instances `ftv38`, `ftv44`, `ftv120`, and `ftv140` appeared to be the most difficult ones to solve and most of the other metaheuristics did not present results for these instances. Another remark concerns the use of suitable parameter settings depending on the specific instance being solved. For example, in the tests carried out by Nagata and Kobayashi as well as by Walters, the population size was optimized as a function of the instance size. We also avoid using the optimal solution value or time as a stopping condition. In the results of Table 4, the same parameters were used for all 27 instances, yet the SAX/RAI memetic algorithm was able to find the optimal solution at least five times per instance from 20 runs. Considering the total of 810 runs in these computational experiments, the method was able to find optimal solutions for 83.5% of the experiments, with an average deviation of 0.04%. Excluding the four hard `ftv` instances mentioned above, the average quality improves to 0.01%, while the percentage of optimal solutions found increases to 91%.

When our method is compared to the local search iKP4 and the *truncated branch-and-bound*, the conclusion is that the proposed method is able to find higher final solutions, expending, however, more time. This suggests that a future research effort should be directed to the development of "complete memetic algorithms", hybridizing once again now by adding a *branch-and-bound*, or other type of exact method, to the population-based search. In our opinion, from all heuristics and metaheuristics presented in Table 4, Helsgaun gives the best tradeoff of *Gap (%)* vs. *CPU Time*.

Table 6 compares the performance of our method using other ATSP instances, generated from STSP instances of the TSPLIB with known optimal solutions. The set of instances is composed by all instances for which the optimal tour is available in the TSPLIB, with the exception of `pr2392`, for which the Java compiler could not allocate memory for the distance matrix.

We use an approach similar to that suggested by Fischetti and Toth (1997) to generate asymmetric instances from symmetric ones. We assume that, given an STSP instance, or an upper diagonal distance matrix, it will be necessary to calculate the lower diagonal distance matrix such that the resulting instance should be asymmetric. Let c_{ij} be the distance between the cities i and j, with $j > i$ (the upper diagonal matrix) and σ the average distance. Hence, for all pairs of cities (j, i), such that $j > i$, if the arc (j, i) is not part of the optimal tour of the symmetric TSP instance, the intercity cost $c_{ji} = c_{ij} + k\sigma$ is calculated, with k a factor uniformly generated at random in the range $[0, k']$. This said, the optimal symmetric tour will be guaranteed to have the same cost as the ATSP instance generated. In this new experiment, two values of k' were used ($k' = 5\%$ and $k' = 50\%$).

Table 6 shows that SAX/RAI provided results similar to those reported in Table 4. For some instances, the quality decreased, and only a few optimal solutions were found for the 20 trials. The asymmetric version of instance brg180 can, for example, be classified as difficult for SAX/RAI MA, possibly due to the low quality of the initial population that was found by

Table 6. Computational results using SAX/RAI on ATSP instances generated from STSP instances with $k' = 5\%$ and $k' = 50\%$.

	$k' = 5\%$					$k' = 50\%$				
Instance	InitGap	Opt/ trials	Gap (%)	Gen	CPU time	InitGap	Opt/ trials	Gap(%)	Gen	CPU time
ulysses16	1.56	20	0.00	92	0.09	3.79	20	0.00	92.00	0.12
ulysses22	6.06	20	0.00	102.8	0.12	5.83	20	0.00	102.10	0.17
gr24	8.49	20	0.00	103.95	0.13	19.89	20	0.00	103.45	0.17
fri26	5.23	19	0.07	105.2	0.15	5.34	20	0.00	103.15	0.17
bayg29	6.58	20	0.00	104.15	0.16	10.68	20	0.00	102.00	0.24
bays29	7.72	20	0.00	104.65	0.18	7.82	20	0.00	102.25	0.20
att48	9.00	19	0.19	117.60	0.40	21.75	20	0.00	103.65	0.36
gr48	4.04	20	0.00	113.60	0.35	19.6	20	0.00	102.80	0.34
eil51	7.28	07	0.67	127.70	0.49	18.08	20	0.00	106.10	0.37
berlin52	14.19	20	0.00	105.65	0.34	9.36	20	0.00	102.20	0.37
st70	12.15	17	0.20	128.45	0.70	24.44	20	0.00	107.30	0.55
eil76	15.43	04	0.92	141.15	0.92	14.68	20	0.00	114.35	0.68
pr76	14.07	20	0.00	111.65	0.52	2.15	20	0.00	103.60	0.59
gr96	12.50	17	0.07	137.75	1.06	12.91	20	0.00	133.50	1.24
kroa100	12.86	18	0.18	117.95	0.79	23.14	20	0.00	107.10	0.83
kroc100	8.20	20	0.00	112.65	0.75	33.51	20	0.00	106.45	0.85
krod100	22.92	13	1.45	131.60	1.07	2.57	20	0.00	106.25	0.85
rd100	21.48	20	0.00	120.15	0.86	12.76	20	0.00	105.10	0.81
eil101	8.90	18	0.11	146.25	1.50	0.00	20	0.00	101.00	0.85
lin105	13.47	20	0.00	110.60	0.83	39.53	20	0.00	104.05	1.00
gr120	13.45	20	0.00	131.50	1.36	6.14	20	0.00	113.85	1.24
ch130	17.89	18	0.42	146.75	1.86	22.24	20	0.00	122.50	1.58
ch150	22.10	20	0.00	125.25	1.66	18.18	20	0.00	109.90	1.52
brg180	50.51	00	14.4	163.30	7.37	188.51	00	33.33	162.80	10.68
gr202	14.95	20	0.00	156.05	4.39	11.69	18	0.29	166.10	6.71
tsp225	16.62	00	1.72	180.00	9.25	23.67	00	4.76	175.45	13.06
a280	20.86	08	0.18	183.20	8.16	22.22	01	0.83	178.55	15.18
pcb442	17.21	19	0.01	192.85	24.66	1.00	20	0.00	144.60	19.97
pa561	20.59	00	3.96	211.00	104.64	7.13	13	0.07	209.65	71.77
gr666	14.13	00	1.97	216.00	211.26	3.79	00	7.40	175.00	233.02
pr1002	19.66	00	6.60	228.50	785.21	12.29	00	4.31	211.20	811.02
Average	14.20	14.7	1.07	137.74	37.78	19.68	16.5	1.65	125.10	38.31

the AP + patching heuristic (188.51%). The generator of the ATSP instances using the STSP instances with known optimal solutions was implemented in Java programming language. The generator, as well as the problem data, is available upon request from the authors.

6. Conclusions

This paper has proposed a new memetic algorithm that uses a new local search, called Recursive Arc Insertion (RAI), for the asymmetric traveling salesman problem (ATSP). This MA implementation introduces several features to improve the performance, including a hierarchically structured population organized as a complete ternary tree of 13 agents clustered in 4 subpopulations, each composed of a leader and 3 supporters agents. Each agent handles two feasible solutions: a *Current* solution used in the operations performed by the method, such as recombination, mutation and local search, and a *Pocket* solution, which is updated from the *Current* solution whenever the latter yields a better quality solution than that already stored in the *Pocket*. The ternary tree is updated in such a way that the root node always handles the best solution found so far. The update mechanism consists of exchanging the *Pocket* solution of the leader with the *Pocket* solution of one of its supporters whenever the latter produces a better quality solution than that stored in the former. Four recombination operators have been tested in conjunction with the RAI local search, and computational results for selecting a suitable operator have shown that the best tradeoff was achieved by Strategic Arc Crossover (SAX).

Computational experiments were conducted on 27 ATSP instances of the TSPLIB (20 runs for each instance). The proposed method was able to find 83.5% of optimal solutions in an average *CPU Time* of 1.07 seconds, with an average solution quality of 0.04%. Tests with another class of asymmetric instances generated from known optimal tours of symmetric instances of the TSPLIB have shown a similar behavior. Moreover, comparisons with six other metaheuristics have demonstrated that the present method outperformed all previous approaches. When comparing our MA with three approximation algorithms, two local search approaches and a truncated branch-and-bound, also considering the asymmetric instances available in the TSPLIB, our MA can be considered competitive.

It seems that the performance of this method could be improved even more if parameters were tuned to the specific set of instances to be used for comparisons but, since robustness is a key feature for method applicability, the parameter settings used here were identical for the whole set of instances and in all tests.

Acknowledgments

This research was supported by the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP, Brazil), the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq, Brazil). We also thank Gilberto Tin Jr., Andréa Toniolo, Cezar Pozzer and Rodrigo Gonçalves for helpful discussions on the use of the Java programming language.

The authors would like to thank two anonymous referees for their valuable comments which have helped improve the quality and readability of the paper.

References

Balas, E. and P. Toth. (1985). "Branch-and Bound Methods." In E.L. Lawler, J.K. Lenstra, A.H.G. Rinnoy Kan, and D.B. Shmoys (eds.), *The Traveling Salesman Problem*. John Wiley and Sons.

Bentley, J.L. (1990). "Experiments on Traveling Salesman Heuristics." In *Proceedings of the First Annual ACM-SIAM Symposium on Computational Geometry*, pp. 91–99.

Berretta, R. and P. Moscato. (1999). "The Number Partitioning Problem: An Open Challenge for Evolutionary Computation." In D. Corne, M. Dorigo, and F. Glover (eds.), *New Ideas in Optimization*, Chapter 17. McGraw-Hill.

Carpaneto, G., M. Dell'Amico, and P. Toth. (1995). "Exact Solution of Large-Scale Asymmetric Traveling Salesman Problems." *ACM Transactions on Mathematical Software* 21(4), 394–409.

Choi, I.-C., S.-I. Kim, and H.-S. Kim. (2003). "A Genetic Algorithm with a Mixed Region Search for the Asymmetric Traveling Salesman Problem." *Computers & Operations Research* 30(5), 773–786.

Cirasella, J., D.S. Johnson, L.A. McGeoch, and W. Zhang. (2001). "The Asymmetric Traveling Salesman Problem: Algorithms, Instance Generators, and Tests." In A.L. Buchsbaum and J. Snoeyink, (eds.), *Proceedings of ALENEX01*, *Springer Lecture Notes in Computer Science* 2153, pp. 32–59.

Fiechter, C.-N. (1994). "A Parallel Tabu Search Algorithm for Large Traveling Salesman Problems." *Discrete Applied Mathematics* 51, 243–267.

Fischetti, M. and P. Toth. (1997). "A Polyhedral Approach to the Asymmetric Traveling Salesman Problem." *Management Science* 43(11), 1520–1536.

Fischetti, M., P. Toth, and D. Vigo. (1994). "A Branch-and Bound Algorithm for the Capacitated Vehicle Routing Problem on Directed Graphs." *Operations Research* 42, 846–859.

França, P.M., A.S. Mendes, and P. Moscato. (2001). "A Memetic Algorithm for the Total Tardiness Single Machine Scheduling Problem." *European Journal of Operational Research* 132, 224–242.

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

Garey, M.R. and D.S. Jonhson. (1979). *Computers and Intractability*: *A Guide to the Theory of NP-Completeness*. Freeman.

Glover, F. (1996). "Finding a Best Traveling Salesman 4-opt Move in the Same Time as a Best 2-opt Move." *J. Heuristics* 2(2), 169–179.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.

Gorges-Schleuter, M. (1989). "ASPARAGOS: An Asynchronous Parallel Genetic Optimization Strategy." In J.D. Schaffer (ed.), *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 422–427.

Gorges-Schleuter, M. (1997). "Asparagos96 and the Traveling Salesman Problem." In T. Baeck, Z. Michalewicz, and X. Yao (eds.), *Proceedings of the IEEE International Conference on Evolutionary Computation*, pp. 171–174.

Gutin, G. and A.P. Punnen. (2002). *The Traveling Salesman Problem and Its Variations*. Kluwer Academic Publishers.

Helsgaun, K. (2000). "An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic." *European Journal of Operational Research* 126(1), 106–130.

Holstein, D. and P. Moscato. (1999). "Memetic Algorithms Using Guided Local Search: A Case Study." In D. Corne, M. Dorigo, and F. Glover (eds.), *New Ideas in Optimization*, Chapter 15 McGraw-Hill.

Johnson, D.S. and L.A. McGeoch. (1997). "The Traveling Salesman Problem: A Case Study." In E. Aarts and J. K. Lenstra (eds.), *Local Search in Combinatorial Optimization*. John Wiley & Sons.

Johnson, D.S., G. Gutin, L.A. McGeoch, A. Yeo, W. Zhang, and A. Zverovich. (2002). "The Asymmetric Traveling Salesman Problem: Algorithms, Instance Generators and Tests." In Gregory Gutin and Abraham P. Punnen (eds.), *The Traveling Salesman Problem and its Variations*, Chapter 10. Kluwer Academic Publishers, pp. 445–488.

Jünger, M., G. Reinelt, and G. Rinaldi. (1995). "The Traveling Salesman Problem." In M. Ball, T. Magnanti, C.L. Monma, and G.L. Nemhauser (eds.), *Handbooks in Operations Research and Management Sciences: Networks*. North-Holland.

Kanellakis, P.C. and C.H. Papadimitriou. (1980). "Local Search for the Asymmetric Traveling Salesman Problem." *Operations Research* 28(5), 1086–1099.

Karp, R.M. (1977). "Probabilistic Analysis of Partitioning Algorithms for the Traveling Salesman Problem in the Plane." *Math. Operational Research* 2, 209–224.

Karp, R.M. (1979). "A Patching Algorithm for the Nonsymmetric Traveling Salesman Problem." *SIAM J. Comput.* 8, 561–573.

Laporte, G. (1992). "The Traveling Salesman Problem: An Overview of Exact and Approximate Algorithms." *European Journal of Operational Research* 59(2), 231–247.

Lin, S. and B.W. Kernighan. (1973). "An Effective Heuristic Algorithm for the Traveling Salesman Problem." *Operations Research* 21(2), 498–516.

Mendes, A.S., F.M. Müller, P.M. França, and P. Moscato. (2002). "Comparing Metaheuristic Approaches for Parallel Machine Scheduling Problems with Sequence Dependent Setup Times." *Production Planning & Control* 13, 143–154.

Merz, P. and B. Freisleben. (1997). "Genetic Local Search for the TSP: New Results." In Proceedings of the 1997 IEEE International Conference on Evolutionary Computation, pp. 159–164.

Merz P. and B. Freisleben. (2001). "Memetic Algorithms for the Traveling Salesman Problem." *Complex Systems* 13(4), 297–345.

Miller, D.L. and J.F. Pekny. (1991). "Exact Solution of Large Asymmetric Traveling Salesman Problems." *Science* 251, 754–761.

Moscato, P. (1989). "On Evolution, Search, Optimization, Genetic Algorithms, and Martial Arts: Towards Memetic Algorithms." Technical Report, Caltech Concurrent Computation Program, C3P Report 826.

Moscato, P. and M.G. Norman. (1992). "A Memetic Approach for the Traveling Salesman Problem. Implementation of a Computational Ecology for Combinatorial Optimization on Message-Passing Systems." In M. Valero, E. Onate, M. Jane, J.L. Larriba, and B. Suarez (eds.), *Parallel Computing and Transputer Applications.* IOS Press, pp. 187–194.

Moscato, P. and F. Tinetti. (1992). "Blending Heuristics with a Population-Based Approach: A Memetic Algorithm for the Traveling Salesman Problem." CeTAD, Report 92-12. Universidad Nacional de La Plata, Argentina.

Moscato, P. (1993). "An Introduction to Population Approaches for Optimization and Hierarchical Objective Functions: A Discussion on the Role of Tabu Search." *Annals of Operations Research* 41, 85–121.

Moscato, P. (1999). "Memetic Algorithms: A Short Introduction." In D. Corne, M. Dorigo, and F. Glover (eds.), *New Ideas in Optimization*. McGraw-Hill.

Nagata, Y. and S. Kobayashi. (1997). "Edge Assembly Crossover: A High-power Genetic Algorithm for the Traveling Salesman Problem." In *Proceedings of the Seventh International Conference on Genetic Algorithms*, pp. 450–457.

Paechter, B., A. Cumming, M.G. Norman, and H. Luchian. (1996). "Extensions to a Memetic Timetabling System." In E.K. Burke and P. Ross (eds.), *The Practice and Theory of Automated Timetabling*, *Lecture Notes in Computer Science*, Vol. 1153. Springer Verlag, pp. 251–265.

Potvin, J.Y. (1993). "The Traveling Salesman Problem: A Neural Network Perspective." *ORSA Journal on Computing* 5, 328–348.

Reinelt, G. (1991). "TSPLIB—A Traveling Salesman Library." *ORSA Journal on Computing* 3, 376–384.

Stützle, T. and M. Dorigo. (1999). "ACO Algorithms for the Traveling Salesman Problem." In K. Miettinen, M. Makela, P. Neittaanmaki, J. Periaux (eds.). *Evolutionary Algorithms in Engineering and Computer Science*. Wiley.

Walters, T. (1998). "Repair and Brood Selection in the Traveling Salesman Problem." In A.E. Eiben, T. Bäch, M. Schoenauer, and H.P. Schwefel (eds.), *Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature*, pp. 813–822.

Whitley, D., T. Starkweather, and D. Shaner. (1991). "The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination." In L. Davis (ed.), *Handbook of Genetic Algorithms*, pp. 350–372.

Zhang, W. (1993). "Truncated Branch-and-Bound: A Case Study on the Asymmetric TSP." In *Proceedings of Spring Symposium on AI and NP-Hard Problems*, pp. 160–166.

Zhang, W. (2000). "Depth-First Branch-and-Bound versus Local Search: A Case Study." In *Proceedings of 17th National Conf. on Artificial Intelligence (AAAI)*, pp. 930–935.