Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2015, Article ID 648035, 9 pages
http://dx.doi.org/10.1155/2015/648035

Research Article

TPA: A Two-Phase Approach Using Simulated Annealing for the Optimization of Census Taker Routes in Mexico

Silvia Gaona[1,2] and David Romero[3,4]

[1]*Universidad Politécnica del Estado de Morelos, 62550 Cuernavaca, MOR, Mexico*
[2]*Universidad Autónoma del Estado de Morelos, 62209 Cuernavaca, MOR, Mexico*
[3]*Instituto de Matemáticas, Universidad Nacional Autónoma de México, 62210 Cuernavaca, MOR, Mexico*
[4]*CETEC, ITESM, 64849 Monterrey, NL, Mexico*

Correspondence should be addressed to Silvia Gaona; sgaona@upemor.edu.mx

Received 4 April 2015; Revised 17 June 2015; Accepted 15 July 2015

Academic Editor: Krishnaiyan Thulasiraman

Censuses in Mexico are taken by the National Institute of Statistics and Geography (INEGI). In this paper a Two-Phase Approach (TPA) to optimize the routes of INEGI's census takers is presented. For each pollster, in the first phase, a route is produced by means of the Simulated Annealing (SA) heuristic, which attempts to minimize the travel distance subject to particular constraints. Whenever the route is unrealizable, it is made realizable in the second phase by constructing a visibility graph for each obstacle and applying Dijkstra's algorithm to determine the shortest path in this graph. A tuning methodology based on the *irace* package was used to determine the parameter values for TPA on a subset of 150 instances provided by INEGI. The practical effectiveness of TPA was assessed on another subset of 1962 instances, comparing its performance with that of the in-use heuristic (INEGI_{H}). The results show that TPA clearly outperforms INEGI_{H}. The average improvement is of 47.11%.

1. Introduction

The National Institute of Statistics and Geography (INEGI, its acronym in Spanish) is the government institution responsible for requesting, processing, analyzing, preserving, and publishing statistical and geographical data that sustain decision making processes in Mexico. These activities are done continuously, require significant public budgets, and involve specific rules and care to ensure the reliability of the generated information. Among its several functions, the planning and execution of official censuses are important.

To carry out censuses and polls, INEGI faces the logistic problem of determining a minimal length travel route through the city blocks of the instance assigned to each census taker or pollster. This problem is subject to two constraints: (a) the route must be done exclusively by accessible roads, and (b) for practical reasons each city block must be completely traversed before passing to the next one.

Presently, INEGI approaches this problem by applying the following rules of thumb (INEGI_{H}) [1]: (a) the first city block to be traversed by the pollster is the one situated in the northwest corner of its assigned instance; (b) each city block must be completely toured, starting from its northwest corner, before going to the next one; (c) once a city block has been toured, the next one is chosen following a zigzag path from north to south (see Figures 1 and 2).

From the combinatorial optimization viewpoint, this problem can be posed as a Minimal Hamiltonian Path Problem (MHPP) with side constraints. Given a graph G with weighted edges, MHPP consists in finding a minimal length path that visits each node of G exactly once. The similarity between MHPP and the well-known Traveling Salesman Problem (TSP) is clear. Hence, in view of the constraint (b) above, INEGI's problem could be modeled as a directed General Traveling Salesman Problem (GTSP), which, in turn, can be transformed in a directed TSP, allowing us to use the standard solution methods for TSP (see [2] and the references therein).

Also, INEGI's problem could be seen as an unpublished member of the family of arc-routing problems, but as its

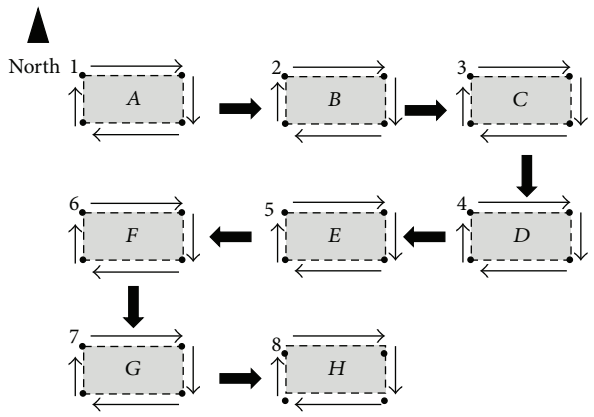

Figure 1: Example of a route produced by INEGI_{H}.

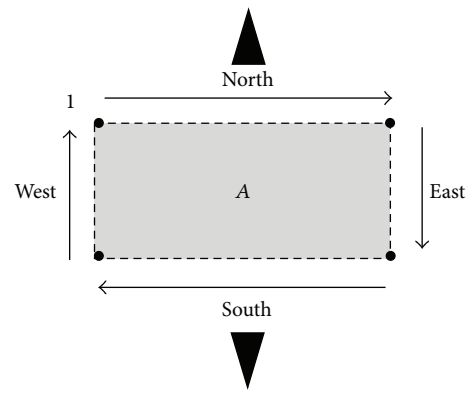

Figure 2: Example of a city block traversal.

instances data are only available in terms of vertex coordinates, this kind of modeling did not appear straightforward. For a thorough account of arc-routing problems see [3].

The decision was to model INEGI's problem as an undirected MHPP with side constraint (b) in a geometric realm. The *GTSP* $\rightarrow$ *TSP* transformation together with standard solution methods did not seem attractive because, being of general use, it does not take advantage of the geometric aspects of the problem, and its performance depends significantly on the number of vertices of the instances. In contrast, this work proposes a solution method whose performance relies more on the number of blocks than on the number of vertices. Other approaches could be the subject of future research. As MHPP has been classified as NP-complete in complexity theory [4], the proposal is an ad hoc heuristic procedure, called TPA (TPA: Two-Phase Approach), composed by two phases. The first one uses an SA algorithm [5] aimed at solving MHPP in a graph whose nodes correspond to the vertices of the polygons defining the city blocks. The second phase is devoted to repair unrealizable solutions (with crossings over the houses or buildings) obtained by the SA algorithm. This is done by means of computational geometry concepts such as visibility graphs [6, 7] and then finding the shortest paths in these graphs with Dijkstra's algorithm [8].

TPA was assessed on a subset of 1962 real-world instances provided by INEGI. Results show that TPA is able to attain an average improvement of as much as 47.11% with respect to INEGI_{H}.

The rest of the paper has been organized as follows. In Section 2 the problem under study is formally defined. The implementation details of TPA are discussed in Section 3. Then, Section 4 presents computational experiments aimed at determining suitable parameter settings for TPA and at comparing its performance with that of INEGI_{H}. Finally, the main conclusions of this research are provided in Section 5.

2. Problem Formalization

The optimization problem studied in this paper consists in finding a minimal length route that traverses all the city blocks in a given instance, subject to the following specific requirements: (a) each city block must be completely traversed before passing to the next one, and (b) the route must be done exclusively by accessible roads, namely, without crossing any city block. In this sense, it can be seen as an instance of the MHPP.

Figure 3(a) shows an unrealizable route because of crossings over city blocks. However, crossings can be eliminated by adding vertices to the route (see Figure 3(b)). Most instances have blocks with irregular, asymmetric geometries, and many more vertices.

More formally, let $B = \{b_1, b_2, \ldots, b_n\}$ be the set of n city blocks of an instance. Each block b_i is defined as an m_i-side polygon with vertex set $V(b_i) = \{v_1, v_2, \ldots, v_{m_i}\}$ and perimeter $P(b_i)$.

Let $V = \bigcup_{b_i \in B} V(b_i)$ be the set composed of all the vertices of n city blocks, and let $w = |V|$. Given a subset $V' \subseteq V$ that includes at least one vertex from each city block $b_i \in B$, a route is an order $r = (v_1, v_2, \ldots, v_l)$ on V' of size $l \geq n$. Then, the total length (cost) of route r is

$$\mathscr{D}(B, r) = \sum_{b_i \in B} P(b_i) + \sum_{1 \leq j < l} \text{dist}\left(v_j, v_{j+1}\right), \tag{1}$$

where $\text{dist}(u, v)$ stands for the Euclidean distance between vertices $u, v \in V'$. The problem consists in finding a route r^* minimizing (1); namely,

$$\mathscr{D}\left(B, r^*\right) = \min\left\{\mathscr{D}(B, r) : r \in \mathscr{R}\right\}, \tag{2}$$

where $\mathscr{R}$ is the set of all the possible routes.

3. TPA

The problem under study was solved by implementing TPA as mentioned in Section 1. Below, its pseudocode is shown as Algorithm 1. In the first phase of TPA a route r^* is constructed by applying the SA algorithm. The core of SA is a neighborhood function, called $N_3(r, u)$, composed of two carefully designed neighborhood relations (see line 10). The route r^* obtained by SA is expected to be optimal or near-optimal; however, it could be unrealizable because, very likely, the neighborhood function generates routes with crossings

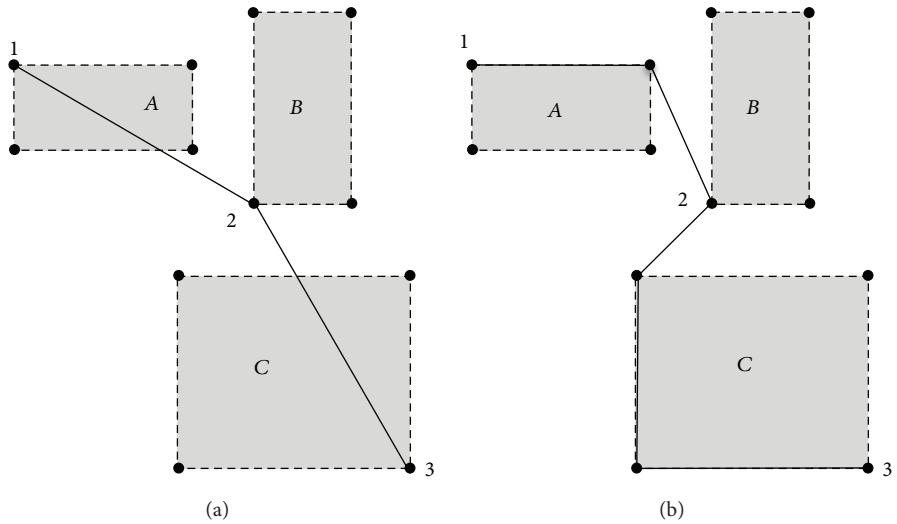

Figure 3: (a) An unrealizable route; (b) a realizable route.

```
(1)  TPA (𝒩, 𝒟, T_i, maxIter, 𝒬)
(2)  begin
(3)    generate a random initial solution r ∈ R
(4)    r* ← r
(5)    T ← T_i
(6)    repeat
(7)      nbIter ← 0
(8)      while nbIter < maxIter do
(9)        nbIter ← nbIter + 1
           /* generate a neighboring solution
              r' ∈ 𝒩(r)   */
(10)       r' ← generateNeighbor(𝒩, r)
(11)       Δ_D ← D(r') − D(r)
(12)       generate a random real number
           μ ∈ [0, 1]
(13)       if (Δ_D < 0) or (e^{−Δ_D/T} > μ) then
(14)             r ← r'
(15)             if D(r') < D(r*) then r* ← r'
(16)       end
(17)     end
(18)     T ← 𝒬(T)
(19)   until stopCondition( )
(20)   r'' ←repairCrossings(r*)
(21)   return r'
(22) end
```

Algorithm 1: TPA algorithm.

over city blocks. Thus, the second phase of TPA (see line 20) is devoted to repair those unrealizable solutions. This is done by adding to r^* one or more vertices of some city blocks so as to avoid crossings over them. The implementation details of the TPA algorithm are as follows.

3.1. Internal Representation. During the SA process, a route r (a solution) is represented by vectors $B = (b_1, b_2 \ldots, b_n)$ and $V = (v_1, v_2, \ldots, v_n)$, where, for $i = 1, \ldots, n$, b_i is the city block visited in the ith step of the route and v_i is the vertex of block b_i where the route leaves b_i to go to block b_{i+1}, or where the route arrives to b_i when coming from block b_{i-1}.

3.2. Neighborhood Function. The main objective of the neighborhood function $N_3(r, u)$ is to produce solutions (routes) by making small changes to the current solution. This neighborhood function is composed of two neighborhood relations (see Algorithm 2). The first one, $N_1(r)$, generates a neighboring solution r' by swapping, in the current solution r, two randomly selected city blocks (uniform distribution). Its range is $n * (n-1)/2$.

The second neighborhood function, $N_2(r)$, changes the vertex of one randomly selected city block, currently used in route r, to produce a neighboring solution r' with the following procedure. First a city block, say b_i, is randomly selected. Then, the next block in the current route r (namely, b_{i+1}) is also considered to identify one of its vertices satisfying the conditions: (a) it minimizes the distance between city blocks b_i and b_{i+1} and (b) it avoids crossings when connecting these two city blocks (see Algorithm 3). The latter condition is ensured by an efficient verification of lines intersection through computational geometry procedures [6]. The range of $N_2(r)$ is bounded by $n * \arg\max_{b_i \in B} |V(b_i)| - 1$.

During the search process, a combination of the $N_1(r)$ and $N_2(r)$ neighborhood relations is employed. The former is applied with probability p, while the latter is employed at $(1 - p)$ rate. Thus, the neighborhood function $N_3(r, u)$ is defined as

$$N_3(r, u) = \begin{cases} N_1(r) & \text{if } u \le p \\ N_2(r) & \text{if } u > p, \end{cases} \tag{3}$$

where u is a random number uniformly distributed in the interval $[0, 1]$.

```
(1)  generateNeighbor (𝒩, r)
     /* r is represented using two vectors, one of blocks B and one of vertices V. B[i]
     represents the ith-block visited in the path and V[i] the vertex used for this block */
(2)  begin
(3)    generate a random integer number μ ∈ [1, 1000]
(4)    if (μ ≤ P_vec) then
(5)        select two different blocks at random positions i and j in r
           /* exchange the blocks B[i] and B[j] in the solution r */
(6)        r' ← swapBlocks (r, i, j)
(7)    else
(8)        select a block at a random position i in r, that is, B[i]
(9)        minDist ← distance(V[i], V[i + 1])
(10)       v* ← V[i + 1]
           /* look for the vertex v* of B[i + 1] which is closer to V[i] and does not generate crossings */
(11)       for every vertex v of block B[i + 1] do
(12)           dist ← distance(v, V[i])
(13)           cross ← verifyCrossings(B[i], V[i], B[i + 1], v)
(14)           if (cross = false) and (dist ≤ minDist) then
(15)               minDist ← dist
(16)               v* ← v
(17)           end
(18)       end
           /* assign the vertex v* as the one used by block B[i + 1] */
(19)       r' ← changeCurrentVertex(r, i + 1, v*)
(20)   end
(21)   return r'
(22) end
```

Algorithm 2: Generate neighboring solution.

```
(1)  verifyCrossings (B[i], V[i], B[i + 1], v)
     /* verifyLineIntersection( ) returns true if the two lines intersect */
(2)  begin
(3)      a ← V[i]
(4)      b ← v
(5)      selfCrossings ← false
         /* verify self crossings */
(6)      for every edge e of block B[i] do
(7)          selfCrossings ← verifyLineIntersection(e, ab)
(8)      end
(9)      otherCrossings ← false
         /* verify crossings with block B[i + 1] */
(10)     for every edge e of block B[i + 1] do
(11)         otherCrossings ← verifyLineIntersection(e, ab)
(12)     end
(13)     return (selfCrossings or otherCrossings)
(14) end
```

Algorithm 3: Verify crossings.

3.3. Evaluation Function. The selection of the evaluation function is a critical aspect of any search procedure. First, to efficiently test each potential solution, the evaluation function must be as simple as possible. Secondly, it must be sensitive enough to identify promising search regions in the solution space. Finally, the evaluation function must be consistent: a solution that is better than others must contribute with a better value of the objective function [9].

In TPA implementation, the cost of a solution r is calculated using formula (1). To determine the cost of a route r with formula (1), the perimeter of every city block $b_i \in B$ and the Euclidean distance between every pair of vertices $(v_i, v_{i+1}) \in r$ in the route must be computed. $O(n + l)$ instructions must be executed (remember that n represents the number of city blocks in the problem instance and l is the size of an order (route) on $V' \subsetneq V$) by this *complete evaluation scheme*.

```
(1)  repairCrossings (r*)
(2)  begin
     /* list of detected crossings in routing r*  */
(3)    C ← ∅
       /* detect crossings in routing r*  */
(4)    for every edge V[i], V[i + 1] used in routing r* do
         /* crossings with block B[i]  */
(5)      C ← selfCross(V[i], V[i + 1], B[i])
         /* get a list of neighbor blocks  */
(6)      NB ← getNeighborBlocks(B[i], B[i + 1])
         /* crossings with neighbor blocks  */
(7)      C ← NeighborCross(V[i], V[i + 1], NB)
(8)    end
(9)    r'' ← ∅
(10)   for every detected crossing in C do
           /* get a list of neighbor blocks  */
(11)       NB ← getNeighborBlocks(B[i], B[i + 1])
                 /* construct a visibility graph  */
(12)       G ← visibilityGraph(V[i], V[i + 1], NB)
               /* find a path without crossings and minimum distance between V[i] and V[i + 1]  */
(13)       P ← findBestPath(G)
           /* repair routing r* using the path P to avoid a particular crossing  */
(14)     r'' ← r'' ∪ repairRouting(r*, P)
(15)   end
       /* return a repaired routing r'' with minimum total distance  */
(16)   return r''
(17) end
```

Algorithm 4: Repair crossings over the blocks.

Nevertheless, TPA employs an *incremental evaluation* of the neighboring solutions which takes advantage of the fact that the evaluation function $\mathcal{D}(B, r)$ contains a constant term, $\sum_{b_i \in B} P(b_i)$. This is precomputed only once and used along the search process. Furthermore, when a neighboring solution r' is generated with function $N_1(r)$, the incremental evaluation scheme only recomputes the (up to four) Euclidean distances that change, in order to obtain the cost of the route r'. This is much faster than the $O(n + l)$ operations originally required.

A similar incremental evaluation mechanism was implemented for the neighborhood function $N_2(r)$. As a consequence, TPA analyzes thousands of neighboring solutions employing only a very small fraction of the time that would be required by the complete evaluation scheme. This is possible thanks to the use of appropriate data structures.

3.4. Cooling Schedule. A cooling schedule is defined by the following parameters: initial temperature, maximum number of solutions generated at each temperature value (Markov chain length), rule for decreasing the temperature, and stop condition. The cooling schedule governs the convergence of any SA [10].

In TPA implementation, a geometrical cooling scheme (static) was defined due to its simplicity, requested computational time, and quality of the solutions produced in preliminary experiments. It starts at an initial temperature T_i. Then, the temperature is decreased at each round by a positive factor $\alpha < 1$ using the relation $T_j = \alpha T_{j-1}$. For each temperature T_j, the maximum number of neighboring solutions was fixed to maxIter $= \mu * (N_1 + N_2)$. Hence, it depends directly on the range of the neighborhood function and on the constant μ, which is called "moves factor." This is because more neighboring solutions are required for denser instances.

3.5. Stop Condition. TPA stops if either the best-known solution does not change during a predefined number of consecutive temperature decrements *maxNI* or the current temperature is lower than a prefixed value T_f.

3.6. Repair Mechanism. Let r denote the route that SA offers as optimal solution in the first phase of TPA. Recall that r is defined by vectors $(b_1, b_2, \ldots, b_n)$ and $(v_1, v_2, \ldots, v_n)$. The second phase of TPA consists in applying the following repair mechanism on r (see Algorithm 4).

Let $h(b)$ denote the centroid of city block b and, for $i = 1, \ldots, n-1$, let $\varphi(i) = 2 * \text{dist}(h(b_i), h(b_{i+1}))$, where $h(b_i)$ and $h(b_{i+1})$ are the centroids of two neighboring blocks. Further, for $i = 1, \ldots, n$, let $\Gamma(i)$ be the set of blocks b in the instance such that

$$\text{dist}\left(h\left(b\right), h\left(b_i\right)\right) \leq \varphi\left(i\right) \quad \text{or}$$
$$\text{dist}\left(h\left(b\right), h\left(b_{i+1}\right)\right) \leq \varphi\left(i\right). \tag{4}$$

Now, let $C(i)$ denote a set of blocks in $\Gamma(i)$ that are crossed by the line segment going from v_i to v_{i+1}, for $i = 1, \ldots, n-1$ (see lines 4–8). A line segment ℓ crosses a city

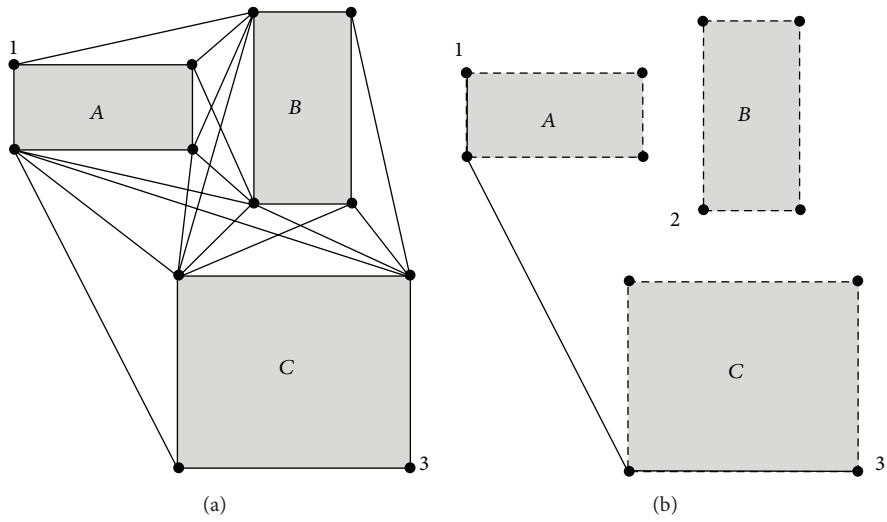

Figure 4: (a) Example of a visibility graph. (b) A realizable path between 1 and 3.

block b, if ℓ has nonempty intersection with the interior of the polygon defining b. The crossings can be of two types: (a) self-crossings, when an edge (v_i, v_{i+1}) crosses the city block b_i itself, and (b) neighboring-crossings, those belonging to $C(i) \setminus b_i$. For example, considering the blocks of Figure 3(a) as a subset of an instance and block A as reference, the line segment 1-2 is a self-crossing, and the segment 2-3 is a neighboring-crossing.

Then, whenever $C(i)$ is not empty, a set of neighboring blocks NB is produced (see lines 10–15). Set NB serves to build up a visibility graph $G = (V, E)$; see [7]. In G, the set of nodes V is composed of the vertices of all city blocks in NB, and edge $(u, v) \in E$ if (u, v) is side of a city block in NB or if the line segment with endpoints u and v does not cross a block in NB (namely, u is visible from v and vice versa). Each edge in E has as weight the Euclidean distance between its endpoints.

Figure 4(a) presents a visibility graph between vertices 1 and 3. Figure 4(b) shows a realizable path between vertices 1 and 3. A shortest path between vertices v_i and v_{i+1} in G is obtained by means of Dijkstra's algorithm. Of course, this path, of minimal length, does not cross any single block. Dijkstra's algorithm is a well-known, efficient algorithm to solve the so-called Shortest Path Problem (SPP) in a graph with nonnegative edge costs [8].

4. Computational Experiments

In this section, two computational experiments are presented. The first one was aimed at determining the parameter values of TPA that yield the best trade-off between solution quality and computing time. The second experiment was carried out to compare TPA with INEGI_{H} in solving the route problem. For these experiments, the algorithms were coded in Java SE Development Kit version 7, update 51. They were run sequentially into a cluster equipped with 10 InfiniBand interconnected nodes, each of which features 8 cores running at 2.66 GHz, has a total of 16 GB of RAM, and uses CentOS distribution of the Linux Operating System.

4.1. TPA Parameter Tuning. Optimizing parameter values is an important activity when implementing efficient algorithms. The literature offers many possibilities to find a combination of parameter values that allow a particular algorithm to furnish the best solution quality in reasonable computational times [11, 12].

In this work, the strategy was to find the most appropriate parameter values of TPA by employing the iterated racing procedure (*I/F-Race*) [13]. This offline automatic configuration procedure is implemented in *R* as part of the *irace* package [14] and has been successfully used in several research projects [15]. *I/F-Race* is based on the use of racing [16] and Friedman's nonparametric two-way analysis of variance by ranks. It consists of the following three steps, executed iteratively until a predefined termination criterion is met: (1) sampling new parameter configurations according to either a normal or a discrete distribution, depending on whether the parameters are numeric or categorical, (2) selecting the best parameter configurations from the newly sampled ones by means of racing, and (3) updating the sampling distribution in order to bias the sampling towards the best configurations. Further details of the *I/F-Race* iterative procedure and its implementation can be found in [13, 14, 16].

For the experiments INEGI provided a set of 2112 instances, where the number n of city blocks varies from 3 to 103, and the total number of vertices w is in the range $[15, 8461]$. From this set a subset of 150 instances served in the tuning experiments, and the remaining 1962 instances were used during the performance evaluation of TPA, as it will be detailed later. From preliminary experiments, it was found that TPA required different parameter configurations depending on the density of the instance to be solved. The *density* of a simple undirected graph $G = (V, E)$ is defined as $2 * |E|/(|V| * (|V| - 1))$. For this specific problem $|V| = w$, and n is the number of blocks. Then, the total number

Table 1: Five ranks of density of the test set with 2112 instances.

Rank	Density		Instances
	From	To	
1	0.00024	0.03207	1859
2	0.03269	0.06456	120
3	0.06555	0.09667	62
4	0.09783	0.12865	33
5	0.13971	0.16190	38
		Total	2112

Table 2: Allowed range of possible parameter settings used for the tuning process.

Rank	T_i	μ	MaxNI
1	[10, 280]	[1, 45]	[50, 500]
2	[10, 220]	[1, 35]	[50, 300]
3	[10, 55]	[1, 25]	[50, 300]
4	[10, 50]	[1, 15]	[50, 300]
5	[10, 30]	[1, 10]	[50, 300]

of blocks sides in a particular instance can be expressed as $|E| = w + n - 1$. Consequently, the density of a particular instance can be computed as

$$\text{density} = \frac{2(w+n-1)}{w(w-1)}. \quad (5)$$

Thus, the decision was to sort the test set obtained from INEGI according to density. Then, five representative ranks of density were identified (see Table 1) and 30 instances from each rank were randomly selected for tuning the TPA algorithm. The idea behind this is to have a representative sample of the instances for this experiment and to produce one good parameter configuration for each density rank.

A sample of 30 statistically representative instances of each rank was used to run the *irace* package [15]. 1500 independent runs were executed for each instance.

In Table 2, the allowed range of possible parameter settings used for the tuning process is presented. T_i, μ, and *maxNI* stand, respectively, for initial temperature, moves factor, and maximum number of consecutive temperature decrements without improvement. For all the ranks, the range of the final temperature T_f was $[0.05, 0.5]$, the range of the cooling factor α was $[0.88, 0.99]$, and the range of the probability of using each of the two neighborhood functions P_{vec} was $[0, 1000]$.

As a result, the five different parameter configurations depicted in Table 2 were obtained. These parameter configurations are therefore used in the experimentation reported next.

In Table 3, the parameter combinations found with *irace* for TPA are presented. T_i grows as density grows. T_f does not present a defined trend; however the values are relatively similar. Moves factor μ grows as density grows; however, in the densest rank, the value drops significantly to 0.66. In this rank, the instances have up to 103 blocks and 8461 vertices. A smaller μ parameter is required because of the large size of the neighborhoods $N_1(r)$ and $N_2(r)$, and the relation maxIter $= \mu * (N_1 + N_2)$.

Table 3: Parameter combinations found with *irace* for TPA.

Rank	T_i	T_f	μ	α	MaxNI	P_{vec}
1	201.18	0.35	0.66	0.99	80	835
2	197.06	0.17	30.55	0.99	267	855
3	53.18	0.25	24.82	0.98	73	851
4	44.62	0.21	13.78	0.99	201	981
5	19.55	0.46	9.15	0.99	297	725

Table 4: Instances used to evaluate TPA performance.

Rank	Instances	Blocks	Vertices	Experiments
1	1829	67167	801167	54870
2	90	689	4566	2700
3	32	160	939	960
4	3	11	57	90
5	8	24	128	240
Total	1962	68051	806857	58860

4.2. Comparing TPA with INEGI$_H$. The main objective of this section is to compare average solutions achieved by TPA with respect to INEGI$_{\text{H}}$. Table 4 shows 1962 (2212 − 150) instances used to evaluate TPA performance. Next, results are compared with INEGI$_{\text{H}}$. Due to the nondeterministic nature of TPA, 30 independent runs were executed for each instance. The parameter values employed by TPA are those previously found in Section 4.1.

The results from this experiment are presented in Table 5, summarizing, for each rank, the average data obtained by the compared algorithms. The solutions found by INEGI$_{\text{H}}$ are listed in column 2. Columns 3 to 6 show, respectively, the best, the average and the standard deviation of the total travel distance (cost) reached by TPA, and the needed CPU time in seconds. Additionally, the average improvement attained by TPA was computed as $100 * (\text{INEGI}_{\text{H}} - \text{Avg.})/\text{INEGI}_{\text{H}}$. Table 5 shows that TPA clearly outperforms INEGI$_{\text{H}}$ with an average improvement of 47.11%. This highlights the suitability of TPA. Also, a relatively small standard deviation is an indicator of the algorithm's precision and robustness. TPA consumes, in average, 0.54 seconds per run. CPU time consumed by TPA is acceptable and fully justified by considering that it is able to outperform INEGI$_{\text{H}}$ in terms of cost. Results achieved by TPA are illustrated in Figure 5. The subset of 1962 instances is indexed in nonincreasing order of cost-improvement.

In regard to the instance with the highest number of blocks (92 blocks, 478 vertices), the improvement was of 70.90%. Further, in the instance with the highest number of vertices (51 blocks, 8461 vertices), the improvement was of 40.30%. The highest improvement, 98.82%, was presented in an instance of 59 blocks and 519 vertices. The lowest improvement was of 1.18% and was presented by an instance with 6 blocks and 275 vertices. TPA seems to be an effective approach to find good quality solutions for the route problem faced by INEGI.

Table 5: Overall performance comparison of TPA with respect to $INEGI_H$.

Rank	$INEGI_H$	TPA Best	TPA Avg.	TPA Std. dev.	TPA Time	Average Improv.
1	1481.70	494.42	798.03	185.84	0.09	46.14%
2	842.89	397.77	407.95	9.92	0.49	51.60%
3	457.90	191.01	197.71	8.98	1.57	56.82%
4	229.07	142.02	142.46	0.54	0.27	37.81%
5	170.03	93.92	97.73	3.12	0.29	43.19%
Avg.	636.31	263.83	328.77	41.68	0.54	47.11%

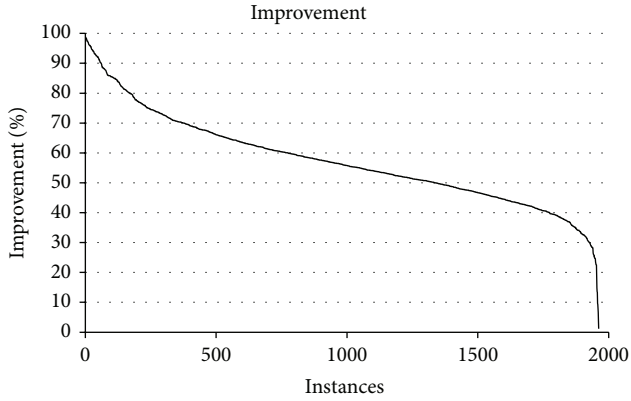

Figure 5: TPA versus $INEGI_H$ for 1962 instances.

In Table 5 it can be observed that ranks 2–5 present a relatively small standard deviation. This is an indicator of the algorithm's precision and robustness, since it shows that, on average, the performance of TPA does not present important fluctuations. However, the standard deviation for rank 1 has a high value, which might be explained by the high variability of the instances' size. For example, the maximum total perimeter of an instance belonging to this subset is 59.48 times the minimum total perimeter of another one. Further, when comparing the length of the optimal route in any two instances of rank 1, the maximum ratio is as much as 256.74.

5. Conclusions and Future Work

In this paper a Two-Phase Approach, called TPA, was proposed for the solution of a logistic problem faced by INEGI. In the first phase of TPA a route r^*, of expected minimal length, is produced by an ad hoc implementation of the SA heuristic. Due to the particular implementation, route r^* is likely to be unrealizable, namely, with crossings over the city blocks. Hence, the second phase of TPA is aimed to repair route r^*, namely, to eliminate the crossings from it. This is done with the use of visibility graphs [6] and Dijkstra's algorithm [8].

The parameter values for TPA were established through intensive experimentation on a representative subset of 150 instances. A tuning methodology based on the *irace* package was applied. The practical effectiveness of TPA was assessed on another subset of 1962 instances drawn from the test set, by comparing its performance with that of the in-use heuristic ($INEGI_H$). The results show that TPA clearly outperforms $INEGI_H$; the average improvement over the total distance traveled is of 47.11% (636.31 versus 328.77).

The problem resolved in this work has common elements with traditional arc-routing problems; however some details make it original. For example, the constraints are that (a) the route must be done exclusively by accessible roads, and (b) for practical reasons each city block must be completely traversed before passing to the next one. Constraint (a) implies a combination of an arc-routing problem with geometric aspects. Standard procedures exclude these considerations and therefore are not applied directly. The composed neighborhood implemented in the SA algorithm is an ad hoc strategy for this specific problem in order to face the presence of blocks and vertices with the challenge of irregular geometries causing unrealizable solutions. Also, the integral solution given to the problem, as a whole, is not reported in the literature. In this sense, this work represents an original contribution.

Although outstanding results were obtained by TPA, they may be still improved. Future work would concentrate on designing alternative neighborhood and evaluation functions, so as to boost the algorithms performance, since it is well-known that these are two important components to define the so-called landscape of the search problem, and they impact thus greatly the overall efficiency [9]. Another future work could focus on a different problem modeling or different heuristic approaches. The instances as well as detailed results of the experiments presented in this paper are available under request.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The first author would like to thank (1) Daniel Rivera, Héctor Sanvicente, and Joaquín Pérez (Ph.D. tutorial committee); (2) PRODEP. This research was partly supported by the Programa para el Desarrollo Profesional Docente (PRODEP), from Mexico, through Grant 103-5/12/4535; (3) Guillermina Urbano and Paulina Baldo from PRODEP; (4) Mireya Gally from Universidad Politécnica del Estado de Morelos.

References

[1] L. E. Martínez-Stiker, Personal communication, INEGI, 2012.

[2] M. Reihaneh and D. Karapetyan, "An efficient hybrid ant colony system for the generalized traveling salesman problem," *Algorithmic Operations Research*, vol. 7, no. 1, pp. 21–28, 2012.

[3] A. Corberán and G. Laporte, Eds., *Arc Routing: Problems, Methods and Applications*, MOS-SIAM Series on Optimization, SIAM, Philadelphia, Pa, USA, 2014.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability. A Guide to the Theory of NP-Completeness*, C.A. Freeman, 1979.

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

[6] J. O'Rourke, *Computational Geometry in C*, Cambridge University Press, New York, NY, USA, 2nd edition, 1998.

[7] M. de Berg, M. van Kreveld, M. Overmars, and S. Otfried, "Visibility graphs," in *Computational Geometry*, chapter 15, pp. 307–317, Springer, New York, NY, USA, 2000.

[8] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[9] E. G. Talbi, *Metaheuristics: From Design to Implementation*, John Wiley & Sons, Hoboken, NJ, USA, 2009.

[10] E. H. Aarts and J. H. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*, John Wiley & Sons, Hoboken, NJ, USA, 1989.

[11] M. Birattari, *Tuning Metaheuristics, A Machine Learning Perspective*, Springer, New York, NY, USA, 2009.

[12] A. Gunawan, H. C. Lau, and Lindawati, "Fine-tuning algorithm parameters using the design of experiments approach," in *Learning and Intelligent Optimization*, vol. 6683 of *Lecture Notes in Computer Science*, pp. 278–292, Springer, Berlin, Germany, 2011.

[13] P. Balaprakash, M. Birattari, and T. Stützle, "Improvement strategies for the F-race algorithm: sampling design and iterative refinement," in *Hybrid Metaheuristics*, vol. 4771 of *Lecture Notes in Computer Science*, pp. 108–122, Springer, Berlin, Germany, 2007.

[14] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari, "The irace package, iterated race for automatic algorithm configuration," Tech. Rep. TR/IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, 2011.

[15] M. López-Ibáñez and T. Stützle, "Automatically improving the anytime behaviour of optimisation algorithms," *European Journal of Operational Research*, vol. 235, no. 3, pp. 569–582, 2014.

[16] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, "A racing algorithm for configuring metaheuristics," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '02)*, San Francisco, Calif, USA, 2002.