Computers & Operations Research 40 (2013) 3218–3229

Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

GRASP with path relinking for the symmetric Euclidean clustered traveling salesman problem

Mário Mestria [1], Luiz Satoru Ochi, Simone de Lima Martins *

Instituto de Computação, Rua Passo da Pátria 156 - Bloco E - 3°andar - São Domingos UFF, Niterói, RJ, CEP: 24210-240, Brazil

ARTICLE INFO

Available online 8 October 2012

Keywords:
Combinatorial optimization
Clustered Traveling Salesman Problem
Heuristics
Path relinking
GRASP

ABSTRACT

In this paper, we propose new heuristics using several path-relinking strategies to solve the Clustered Traveling Salesman Problem (CTSP). The CTSP is a generalization of the Traveling Salesman Problem (TSP) in which the set of vertices is partitioned into clusters and the objective is to find a minimum cost Hamiltonian cycle such that the vertices of each cluster are visited continuously. A comparison among the performance of the several different adopted path-relinking strategies is presented using instances with up to 2000 vertices and clusters varying between 4 and 150 vertices. Also computational experiments were performed to compare the performance of the proposed heuristics with an exact algorithm and a Genetic Algorithm. The obtained computational results showed that the proposed heuristics were able to obtain competitive results related to the quality of the solutions and computational execution time.

1. Introduction

The Traveling Salesman Problem (TSP) is one of most studied problem in the optimization research area. An extension of the TSP, called the Clustered Traveling Salesman Problem (CTSP), is the focus of this paper. This variant was defined by Chisman [1] for modeling a real-world warehousing problem. The problem is stated as follows: A salesman should visit each city once and only once, but some subsets (clusters) of these cities must be visited contiguously.

The applications for the CTSP can be found in automated warehouse routing [1], emergency vehicle dispatching [2], production planning [3], computer disks defragmentation, manufacturing, vehicle routing [4], commercial transactions with supermarkets, shops and grocery suppliers [5].

Most of algorithms already developed for the CTSP consider that the order of visiting the clusters is specified a priori [6–9]. However, in real world applications, usually the order to visit the clusters is not predefined. But few solutions were proposed to solve the CTSP without a pre-order.

For the CTSP, several α-approximation algorithms were developed with different performance guarantees [6,7,10–12], which adapt the algorithms developed in [13,14] developed for the TSP. Most of the approximation algorithms must establish, in each cluster, the starting and ending vertices or/and a prespecified order of visiting the clusters.

An algorithm which provides lower bounds on the optimal tour lengths using Lagrangean relaxation was developed in [15] for the CTSP. The method for obtaining lower bounds was based on the 1-tree relaxation. The computational results are reported for a set of test instances with vertices numbers varying from 80 to 150 and different sizes of clusters. The algorithm obtained an average *gap* equal to 0.35%.

In [7] heuristics were proposed to solve the CTSP with a prespecified order on the clusters. An approximation algorithm with good empirical performance was developed. Additionally, three heuristics were proposed and compared among them. Computational results showed that the best results were obtained by the heuristic, which first transforms the CTSP into a TSP and then apply the GENIUS heuristic.

A Tabu Search heuristic combined with a phase of diversification using a Genetic Algorithm (GA) was proposed in [8] to solve the CTSP with a prespecified order of visiting the clusters. The computational results presented show that Tabu Search outperforms the GA from [9], which exploits order-based crossover operators and local search heuristics. When comparing Tabu Search with a post-optimization procedure [7], Tabu Search obtained results which presented better quality, but demanded more computational time.

In [16] a GA was developed for the CTSP, which first finds inter-cluster paths and then intra-cluster paths. Comparisons of the GA were made with a GENIUS heuristic [7] and lower bounds obtained in [15]. This GA solved problems with up to 500 vertices and 4 and 10 clusters obtaining results within 5.5% of the lower bound.

* Corresponding author.
E-mail addresses: mmestria@ifes.edu.br, mmestria@ic.uff.br (M. Mestria), satoru@ic.uff.br (L. Satoru Ochi), simone@ic.uff.br (S. de Lima Martins).
URL: http://www.ic.uff.br/~satoru/ (L. Satoru Ochi).
[1] Colatina Campus, IFES, Colatina, ES, CEP: 29700-558, Brazil.

0305-0548/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cor.2012.10.001

Another GA was proposed in [17] for the CTSP. The algorithm was implemented using a two-level Genetic Algorithm (TLGA). At the lower level, TLGA finds Hamiltonian cycles in each cluster. At the higher level, the algorithm randomly chooses an edge to be deleted on the cycle in each cluster and simultaneously determines routing among clusters. Computational results demonstrated that TLGA was more effective and efficient than the other Genetic Algorithm developed by the same authors.

Approximate solutions of high-quality have been obtained by using the GRASP metaheuristic to solve combinatorial optimization problems [18,19]. Therefore, in this work, we developed algorithms based on the GRASP metaheuristic [20] for the CTSP without a prespecified order for visiting the clusters, which is more useful for real-world problems and less tackled by works in the literature. We address the issue of designing and verifying procedures for solving the CTSP by developing one traditional GRASP and five heuristics using GRASP with different path-relinking strategies [21,22].

We conducted several computational experiments to compare the performance of all several different adopted path-relinking strategies. We also present a comparison between the performance of the proposed heuristics and an exact algorithm and the Genetic Algorithm developed in [17].

This paper is organized as follows. The problem description and a mathematical formulation are given in Section 2. The heuristics proposed for the CTSP are described in Section 3. Computational experiments performed to compare the results obtained by an exact algorithm and the developed heuristics and among the proposed heuristics are shown in Section 4. A comparison between the proposed heuristics and a Genetic Algorithm developed in [17] is described in Section 5. Conclusions are presented in Section 6.

2. The Clustered Traveling Salesman Problem

The CTSP can be stated as follows: let $G=(V,E)$ be a symmetric complete undirected graph with vertex set $V=\{v_1,v_2,\ldots,v_n\}$ and edge set $E=\{(v_i,v_j): v_i, v_j \in V, i\neq j\}$, and edge weights satisfying the triangle inequality. The vertex set V is partitioned into disjoint clusters $V_1,V_2,\ldots,V_m$ and a non-negative cost c_{ij} is associated to each edge $(v_i,v_j)\in E$. The objective of the CTSP is to find a minimum cost Hamiltonian tour on G, where all vertices of each cluster must be visited contiguously. When there is only a subset $V_i=V$, the CTSP reduces to TSP. Therefore, the CTSP is NP-hard [12].

In this work, the cost c_{ij} represents the Euclidean distance [23] between two vertices v_i and v_j. Because the distances are Euclidean and symmetric, the problem is named as Symmetric Euclidean Clustered Traveling Salesman Problem (SECTSP).

A feasible solution for an example of an instance of the SECTSP is showed in Fig. 1, where the solution consists of a Hamiltonian cycle such that the vertices of each cluster are visited contiguously. The dotted line edges (3, 14), (4, 15), (10, 8), and (16, 12) show the connections inter-cluster and the full line edges the intra-cluster connections.

A formulation for the SECTSP using integer programming is described in [1,24]. The salesman leaves an origin city v_1 and returns to v_1. The cost c_{ij} represents the Euclidean distance between city v_i and city v_j. The salesman proceeds from city v_i to city v_j if and only if $x_{ij}=1$. The formulation is as follows:

$$\text{Minimize } z=\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \tag{1}$$

subject to

$$\sum_{j=1}^{n} x_{ij}=1, \quad \forall i\in V, \tag{2}$$

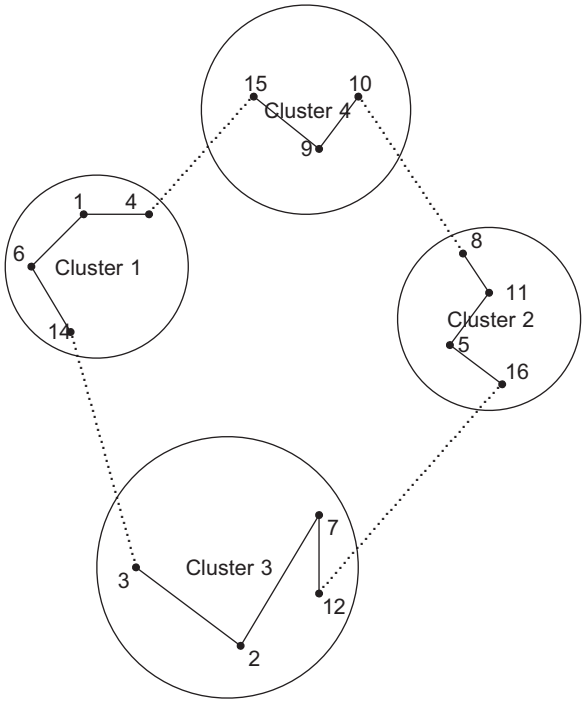

Fig. 1. A feasible solution for an example of an instance of the SECTSP.

$$\sum_{i=1}^{n} x_{ij}=1, \quad \forall j\in V, \tag{3}$$

$$u_i-u_j+(n-1)x_{ij}\leq(n-2), \quad 2\leq i\neq j\leq n, \tag{4}$$

$$\sum_{i\in V_k}\sum_{j\in V_k} x_{ij}=|V_k|-1, \quad \forall V_k\subset V, \quad |V_k|\geq 1, \quad k=1,\ldots,m, \tag{5}$$

$$u_i\geq 0, \quad 2\leq i\leq n, \tag{6}$$

$$x_{ij}\in\{0,1\}, \quad \forall i,j\in V. \tag{7}$$

The objective function (1) minimizes the total distance traveled by the salesman and constraints (2) and (3) ensure that each city is visited once. Constraints (4) serve to eliminate tours that do not begin and end at city v_1 and tours that visit more than $(n-1)$ cities. Constraints (5) state that a Hamiltonian path of length $|V_k|-1$ must go through the $|V_k|$ points of cluster k. Constraints (6) and (7) define the domain of variables.

3. GRASP heuristics for SECTSP

GRASP [25] is a metaheuristic already applied successfully to many optimization problems [18,19]. It consists of a two-phase iterative process. The first phase of a GRASP iteration is the construction phase, in which a greedy randomized solution is built. Since this solution is not guaranteed to be locally optimal, a local search is performed in the second phase. This iterative process is repeated until a termination criterion is met and the best solution found over all iterations is taken as the result.

3.1. Construction phase

The GRASP construction phase developed for the SECTSP applies a penalization for inter-clusters edges which transforms

the SECTSP into a TSP as proposed in [1]. Then an algorithm based on the nearest neighbor heuristic [26] is applied to solve the TSP.

Algorithm 1 describes the construction phase procedure. In line 1, the penalization of the inter-cluster edges is carried out. Then, in line 2, a vertex v_l is randomly chosen and two nearest neighborhood vertices (v_{l+1} and v_{l+2}) are selected to be the initial solution vertices. In line 3, solution S is constructed using these three vertices. The candidate set C is initialized in line 4. In line 6, the k vertices of C which are nearer to the vertices of the partial constructed tour are found and in line 7 the incremental costs obtained when inserting each vertex are evaluated. In lines 8 and 9, the minimum ic_{min} and maximum ic_{max} incremental costs are obtained. The Restricted Candidate List (RCL) is created in line 10 and in line 11, a vertex s is randomly selected from RCL. In line 12, the partial tour (*solution S*) is updated by inserting the vertex s between the two adjacent vertices which results in the minimum cost solution. In line 13, the candidate set C is updated and finally, in line 15, the final solution S is returned.

Algorithm 1. Construction_Phase(α)

```
1:  Penalize_Edges();
2:  IV ← Select_Three_Vertices;
3:  S ← Obtain_Initial_Solution(IV);
4:  C ← V\IV;
5:  while C ≠ ∅ do
6:      NV ← Find_Nearest_Vertices(S);
7:      IC ← Evaluate_Incremental_Costs(NV);
8:      ic_min ← min{ic ∈ IC};
9:      ic_max ← max{ic ∈ IC};
10:     RCL ← {nv ∈ NV | ic(nv) ≤ ic_min + α*(ic_max − ic_min)};
11:     s ← Select_Element_RCL(RCL);
12:     S ← Obtain_Min_Cost_Solution(S,s);
13:     C ← C\{s};
14: endwhile
15: return S
```

In the constructive algorithm described before, a Restricted Candidate List (RCL) is created using a parameter α to restrict the size of this list. The values for α adopted in the constructive heuristics are set using a reactive strategy, which usually leads to better performance than using fixed values and there is no need to perform the tuning of the α parameter values [27]. Reactive strategies consist in selecting α values using a probability based on the costs obtained in the previous GRASP iterations. Initially all α values have the same probability of being selected. As the GRASP iterations are performed, the probability values are adjusted, so that α values which obtained better costs are given higher probability to be selected.

The reactive strategy implemented in this work is based on [27]. Let $\Psi = \{\alpha_1, \ldots, \alpha_m\}$ be the finite set of m possible values for α and let p_i be the corresponding probability of selecting α_i, $i = 1, \ldots, m$. Initially, p_i is uniformly distributed:

$$p_i = 1/m, \quad i = 1, \ldots, m. \tag{8}$$

After T iterations, the p_i values are reevaluated as follows. Let z^* be the best cost solution found in the T previous GRASP iterations and let $\overline{z_i}$ be the average cost solutions obtained using $\alpha = \alpha_i, i = 1, \ldots, m$ during T GRASP iterations. The probabilities are updated after T iterations according to:

$$p_i = q_i \Big/ \sum_{j=1}^{m} q_j, \quad i = 1, \ldots, m, \tag{9}$$

where $q_i = z^*/\overline{z_i}$.

Algorithm 2. *Local_Search_Phase (S)*

```
1:  S* ← S;
2:  repeat
3:    Improve ← false;
4:    for (i = 1; i ≤ m−1; i++) do
5:      for (j = i+1; j ≤ m; j++) do
6:        if (e_i is not adjacent to e_j) then
7:          if (e_i AND e_j are on the same cluster) OR (e_i AND e_j
      are inter-cluster edges) then
8:            S' ← Obtain_New_Solution (S,e_i,e_j);
9:            if f(S') < f(S*) then
10:             Improve ← true;
11:             S* ← S';
12:           end if
13:         end if
14:       end if
15:     end for
16:   end for
17: until Improve = false;
18: return S*
```

3.2. Local search phase

The GRASP local search phase uses the 2-Opt heuristic for exchanging intra-cluster or inter-cluster edges. Algorithm 2 shows the local search procedure. The input parameter is an initial solution S with m edges obtained by the construction procedure. From lines 4 to 16, all combinations of two non-adjacent edges e_i and e_j of S are evaluated. If both edges are in the same cluster or are inter-cluster edges, they are exchanged with two other edges and a new solution is obtained (line 8). If a new solution with a better cost is found, then the incumbent solution is updated in line 11 and a local search is performed with this new solution. This procedure is executed until no better cost solution is found (line 17).

3.3. GRASP and path-relinking strategies

Algorithm 3 presents the GRASP heuristic developed for SECTSP. In line 1, the number of iterations to adjust probabilities for selecting α values is initialized and in line 2 the initial probabilities for selecting α values are set according to Eq. (8). From lines 4 to 17, *maxiter* iterations are performed. In line 5, an α value is selected according to the adjusted probabilities and in line 6 a solution is constructed by Algorithm 1. The local search procedure is executed in line 7 and the best solution found is updated in line 9. The probability associated to α values are periodically updated in line 12, calculated by Eq. (9) as previously described.

Algorithm 3. *GRASP_SECTSP*

```
1:  iterprob ← 1;
2:  initializealphas();
3:  iter ← 1;
4:  while iter <= maxiter do
5:    α ← selectalphas();
6:    S' ← Construction_Phase(α);
7:    S' ← Local_Search_Phase (S');
8:    if f(S') < f(S*) then
9:      S* ← S';
10:   end if
11:   if iterprob = maxprob then
12:     updatealphas();
```

```
13:       iterprob ← 0;
14:     end if
15:     iterprob ← iterprob + 1;
16:     iter ← iter + 1;
17:   endwhile
18:   return S*;
```

Laguna and Martí [28] were the first to use path-relinking within a GRASP strategy. Several extensions, improvements and successful applications of this technique can be found in the literature [29].

Path-relinking generates new solutions by exploring trajectories connecting an initial solution s_i to a guiding solution s_g. The path-relinking procedure consists in selecting moves that introduce attributes contained in the guiding solution s_g to the initial solution s_i until the initial solution is completely transformed in the guiding solution s_g.

To use path-relinking within a GRASP procedure, an elite set *ES* is maintained, in which good solutions found in the previous GRASP iterations are stored.

Two basic strategies for introducing path-relinking into GRASP may be used as described in [29]:

- performing path-relinking after each GRASP iteration using a solution from the elite set and a local optimum obtained after the GRASP local search;
- applying path-relinking to all pairs of elite solutions, either periodically or after all GRASP iterations terminate.

Also, there are several ways to explore the paths between the initial solution and the guiding solution: backward relinking, forward relinking, backward-and-forward relinking, periodical relinking, randomized relinking and truncated relinking.

One important issue in implementing a path-relinking technique is the strategy to construct the elite set. We adopted a fixed size elite set (*ES*) and a solution S' generated by a GRASP iteration is inserted in the *ES* as follows.

A solution S' is always inserted in *ES* if it is not full. Otherwise, the generated solution S' is inserted in *ES* only if its cost is better than the worst cost solution found in *ES* and the difference $dif(S',S_j)$ between S' and all solutions $S_j, j \in ES$ is less than *mindif*. The difference $dif(S_i,S_j)$ between two solutions (S_i and S_j) is calculated by the number of edges needed to be inserted in S_i to transform it into S_j. If the generated solution presents these two conditions, then a solution from *ES* is selected to be replaced by the solution S'. The solution S_d to be deleted from *ES* is the one which presents the smallest $dif(S',S_d)$ value and which cost is worse than the cost of S'.

Algorithm 4. *Elite_Set*(S')

```
1:  if ES is not full then
2:    ES ← ES ∪ {S'};
3:  else
4:    if f(S') < f(S_worstcost) then
5:      count ← 0;
6:      for j = 1 to maxelite do
7:        if dif(S',S_j) > mindif then
8:          count ← count + 1;
9:        end if
10:     end for
11:     if count = maxelite then
12:       valuedif ← dif(S',S_worstcost);
13:       S_out ← S_worstcost;
14:       for j = 1 to maxelite do
15:         if (f(S') < f(S_j)) and (dif(S',S_j) < valuedif) then
16:           valuedif ← dif(S',S_j);
17:           S_out ← S_j;
18:         end if
19:       end for
20:       ES ← ES \ {S_out};
21:       ES ← ES ∪ {S'};
22:     end if
23:   end if
24: end if
25: return ES;
```

Algorithm 4 shows the pseudo-code for the algorithm to construct and maintain the elite set *ES* of size *maxelite*. If the elite set (*ES*) is not full, the solution S' is inserted in *ES* in line 2. Otherwise, the cost of S' is compared to the cost of the worst cost solution in *ES* ($S_{worstcost}$) in line 4. If its cost is worst, then the procedure terminates and the solution S' is not inserted in *ES*. Otherwise, from lines 6 to 10, the procedure verifies if S' differs from all *ES* solutions by *mindif*. If this happens, from lines 11 to 22 an *ES* solution is selected to be deleted from *ES* and the solution S' is inserted in *ES*.

Algorithm 5. *Path_Relinking* (S_1,S_2)

```
1:  if f(S_1) < f(S_2) then
2:    S_i ← S_1;
3:    S_g ← S_2;
4:  else
5:    S_i ← S_2;
6:    S_g ← S_1;
7:  end if
8:  S* ← S_i;
9:  S_int ← S_i;
10: if Symmetrical (S_i,S_g) then
11:   return S_*
12: end if
13: improve ← true;
14: while improve do
15:   improve ← false;
16:   for (j = 1; j ≤ n; j++) do
17:     if e_j(S_int) ≠ e_j(S_g) then
18:       if e_j(S_int) AND e_j(S_g) are on the same cluster then
19:         v_k ← suc(v_j,S_g);
20 :        S' ← New_Solution(S_int,e_j(S_g),v_{j+1},v_k);
21:         if f(S') < f(S*) then
22:           S* ← S';
23:           improve ← true;
24:         end if
25:       end if
26:     end if
27:   end for
28:   S_int ← S*;
29: endwhile
30: return S*
```

Algorithm 5 shows the path relinking procedure developed for SECTSP. The input parameters are two solutions S_1 and S_2. The algorithm returns the best solution (S^*) found in the path between S_1 and S_2. The path relinking procedure is applied by setting the initial solution S_i as the solution which presents the best cost between S_1 and S_2 and S_g as the other one. This strategy was adopted because many works have already showed that usually it

brings the best performance [29]. The path-relinking procedure also considers the orientation of the tour for insertion of edges and vertices. The path-relinking is applied using two different solutions. Two solutions S_i and S_g are considered different solutions if they have different costs and different edges or if they have the same cost and edges but different orientations on the tour defined by the solution.

Let S_i be the initial solution, S_g be the guiding solution, S^* be the best solution, and S_{int} an intermediate solution. For a specific oriented tour t defined by solution S and for any vertex v_j on the tour t, let v_{j-1} be its predecessor and v_{j+1} its successor. The edge connecting vertex v_j to its successor vertex v_{j+1} in tour t defined by solution S is denoted by $e_j(S)$. From lines 1 to 9 the initial, guiding, best and intermediate solutions are initialized.

In line 10, the procedure *Symmetrical* (S_i,S_g) verifies if solutions S_i and S_g are different, by checking if they have different costs and different edges or if they have same costs and edges but different orientations. If solutions S_i and S_g are equal then the path-relinking procedure terminates.

Otherwise, from lines 16 to 27, all vertices v_j from the current intermediate solution S_{int} are evaluated to generate new intermediate solutions. In lines 17 and 18, the procedure verifies if the edges leaving v_j are different in solutions S_{int} and S_g and are intra-cluster edges. If both conditions are met, the procedure continues, otherwise no intermediate solution is generated.

In line 19, the successor v_k of v_j in tour t of the guiding solution S_g is determined. In the next line, the procedure *New_Solution* (S_{int},$e_j(S_g)$,v_{j+1},v_k) obtains a solution S' by inserting $e_j(S_g)$ in S_{int}. To generate the new intermediate solution, it disconnects the vertex v_{j+1} and the vertex v_k from all vertices in S_{int} and connects the vertex v_j to the vertex v_k by the edge $e_j(S_g)$ and connects the vertex v_k to the vertex v_{j+2}. Then it generates a new tour by connecting vertex v_{j+1} between the vertex v_{k-1} and the vertex v_{k+1} in S_{int}.

If no intermediate solution improves the cost of the previous solution, then the path relinking procedure terminates.

We propose in this work to incorporate four different path relinking (PR) strategies to the basic GRASP heuristic by defining distinct ways to perform path relinking. The first developed path-relinking strategy (PR1) consists in applying path-relinking after executing all GRASP iterations. Algorithm 6 presents the GRASP incorporated with this strategy. In line 15, the procedure *Elite_Set* is applied to the solution generated in a GRASP iteration to check if the solution should be inserted in *ES*. After all GRASP iterations are performed, in line 19, the path-relinking is executed for each solution pair of *ES* and each new solution generated by path-relinking is processed by the *Elite_Set* procedure. The path-relinking procedure terminates when *ES* is not updated. The best solution is updated in line 21.

Algorithm 6. *GRASP_with_PR1*

```
1:  iterprob ← 1;
2:  initializealphas();
3:  iter ← 1;
4:  while iter < = maxiter do
5:    α ← selectalphas();
6:    S′ ← Construction_Phase(α);
7:    S′ ← Local_Search_Phase (S′);
8:    if f(S′) < f(S*) then
9:      S* ← S′;
10:   end if
11:   if iterprob = maxprob then
12:     updatealphas();
13:     iterprob ← 0;
14:   end if
15:   ES ← Elite_Set(S′);
16:   iterprob ← iterprob + 1;
17:   iter ← iter + 1;
18: end while
19: S″ ← Pos_Path_Relinking(ES);
20: if f(S″) < f(S*) then
21:   S* ← S″;
22: end if
23: return S*;
```

The second strategy (PR2) is performed in each GRASP iteration between the generated solution after the local search and one solution from *ES* and is shown in Algorithm 7. If the elite size is completely full, a solution is randomly selected from *ES* in line 9 an in line 10 the path-relinking is executed between this selected solution and the solution generated in the GRASP iteration, performing Algorithm 5. The incumbent solution is updated in line 13, if a better solution is found by path-relinking. In line 19, the incumbent solution is processed by the *Elite_Set* procedure.

The third strategy (PR3) consists in performing path-relinking when the elite set *ES* is completely renovated. When *ES* becomes full, this strategy performs a path-relinking among all solutions of *ES*. Then at each iteration, it checks if all solutions of *ES* were replaced by new ones. If this happens, a path-relinking among all solutions of *ES* is performed. Each new solution generated by the path-relinking is processed by Algorithm 4 to check if it should be inserted in *ES*. The fourth strategy (PR4) is similar to PR3, but the path-relinking among all solutions of *ES* is activated every time that one new solution is inserted in *ES*.

Algorithm 7. *GRASP_with_PR2*

```
1:  iterprob ← 1;
2:  initializealphas();
3:  iter ← 1;
4:  while iter < = maxiter do
5:    α ← selectalphas();
6:    S′ ← Construction_Phase(α);
7:    S′ ← Local_Search_Phase (S′);
8:    if iter > |ES| then
9:      S″ ← Sel_Sol_ES(ES);
10:     S′ ← Path_Relinking(S″,S′);
11:   end if
12:   if f(S′) < f(S*) then
13:     S* ← S′;
14:   end if
15:   if iterprob = maxprob then
16:     updatealphas();
17:     iterprob ← 0;
18:   end if
19:   ES ← Elite_Set(S′);
20:   iterprob ← iterprob + 1;
21:   iter ← iter + 1;
22: end while
23: return S*;
```

We developed six GRASP heuristics to solve the SECTSP as follows:

- G: Traditional GRASP as shown in Algorithm 3
- GPR1: GRASP with Path Relinking PR1
- GPR2: GRASP with Path Relinking PR2
- GPR1R2: GRASP with Path Relinking PR1 and PR2

- GPR3: GRASP with Path Relinking PR3
- GPR4: GRASP with Path Relinking PR4.

4. Computational results

There are no instances available for the SECTSP. Therefore, we generated a quite variable set of test instances for the generic version of SECTSP, i.e., without a prespecified order of visiting the clusters to evaluate the heuristics proposed in this work.

Six distinct types of instances were generated: (type 1): instances adapted from the TSPLIB [23] using a k-means clustering algorithm to generate the clusters; (type 2): instances adapted from instances found in the literature for the TSP creating the clusters by grouping the vertices in geometric centers [30]; (type 3): instances generated by using the Concorde interface available in [31]; (type 4): instances generated using the layout proposed in [8]; (type 5): instances similar to type 2, but generated with different parameters; (type 6): instances adapted from the TSPLIB [23], where the rectangular floor plan is divided into several quadrilaterals and each quadrilateral corresponds to a cluster.

All instances are Euclidean instances and they are available via http accessing on: ⟨http://labic.ic.uff.br/Instance/index.php⟩.

4.1. Results obtained by CPLEX

We used the mathematical formulation presented in Section 2 and the software ILOG Parallel CPLEX (version 11.2) [32] to obtain optimal values or lower bound values for these instances. The CPLEX was executed on a 2.83 GHz Intel Core 2 Quad with 4 cores and 8 Gbytes of RAM running the Ubuntu Linux operating system (version 4.3.2-1).

In Table 1, we show the computational results obtained by the software CPLEX for some instances. The first column shows the instance name, followed by an identifier, the number of vertices, number of clusters and the instance type. The last column (%) shows the gap_{lb} value calculated by (10). We set a CPU time limit equal to 25 000 s for all CPLEX executions. The optimal solution was found only for instance I_3 in 442 s. All instances were executed using 25 000 s, therefore, for these instances, CPLEX was aborted before it reached the optimal solution.

The gap_{lb} value was calculated as follows:

$$gap_{lb} = 100 * \left(\frac{best - lb}{best + \epsilon}\right), \tag{10}$$

where *best* is the *best value* found by CPLEX, *lb* the *lower bound*, and ϵ is equal to 10^{-10}.

4.2. Results obtained by the proposed GRASP heuristics

The proposed heuristics were also implemented in the C programming language and executed on the same computer previously described.

All parameter values were set empirically after preliminary experiments, using the instances generated and presented in this work. The value of the penalization for the inter-cluster edges used in the constructive heuristic was set to 10*max$\{c_{ij}\}$, where max$\{c_{ij}\}$ is the maximum cost among all costs of the edges (v_i, v_j).

Ten values were used for $\alpha = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$, which were initially set to have the same probability of being chosen. The number of iterations *maxprob* used as the period to update the α values probability distribution was set to 25. The size of the elite set used for path-relinking was set to 10, and the minimum difference (*mindif*) between a solution candidate to be inserted in the elite set to all solutions in the elite set was set to 5. The maximum number *maxiter* of iterations was set to 200. For all instances, each heuristic was executed 10 times.

Table 2 shows gap_{lb} values for the best results found in 10 executions of GRASP heuristics. The gap_{lb} value was calculated as shown in (10), where *best* is the *best value* found by the heuristic. The first column shows the instance identifiers and the second one presents the value for gap_{lb} calculated for the results obtained by CPLEX. The last six columns show the values gap_{lb} obtained by the GRASP heuristics. The bold values in Table 2 indicate the best values. The last line presents the average of gap_{lb} values. CPLEX obtained three best results, GPR1R2 obtained four, GPR1 got two, and GPR3 and GPR4 obtained one. So, GPR1R2 presented the best performance related to the quality of best solutions.

Table 3 shows the gap_{lb} values for the average results obtained in 10 executions of GRASP heuristics. The gap_{lb} value is calculated as shown in (10), where *best* is the average value found by the heuristic. The first column shows the instance identifiers, followed by the gap_{lb} value calculated for the results obtained by CPLEX. The last six columns show the average values for gap_{lb} obtained by the GRASP heuristics. The bold values in Table 3 indicate the best average values. The last line presents the average of gap_{lb} values. Four best results were obtained by CPLEX and the other eight were obtained by GPR1R2. We can observe that the best average for gap_{lb} values is obtained by GPR1R2 and also that the values obtained by the other heuristics are small and close to the values obtained by CPLEX.

Table 1
Instances with their identifiers (Id.), number of vertices, number of clusters (V_k), type, and CPLEX solutions.

Instances	Id.	# Vertices	# V_k	Type	CPLEX		
					Cost	Lower bound	(%)
10-lin318	I_1	318	10	1	531 931	526 559.70	1.01
10-pcb442	I_2	442	10	1	546 157	536 478.37	1.77
25-eil101	I_3	101	25	6	23 671	23 668.63	**0.01**
144-rat783	I_4	783	144	6	916 103	913 715.52	0.26
300-20-111	I_5	300	20	5	310 590	308 627.83	0.63
500-25-308	I_6	500	25	5	367 509	364 114.62	0.92
300-6	I_7	300	6	3	8956	8916.41	0.44
700-20	I_8	700	20	3	41 615	41 274.00	0.82
C1k.0	I_9	1000	10	2	133 638 625	131 360 206.30	1.70
C1k.1	I_{10}	1000	10	2	130 563 491	128 540 131.50	1.55
200-4-h	I_{11}	200	4	4	62 835	62 391.08	0.71
600-8-z	I_{12}	600	8	4	132 767	128 083.87	3.53
						Average gap_{li}=	1.11

Table 2
Best results for GRASP heuristics executing 200 iterations.

Id.	CPLEX	Best solution values					
		G	GPR1	GPR2	GPR1R2	GPR3	GPR4
I_1	1.01	1.14	0.80	1.02	**0.79**	0.85	0.83
I_2	1.77	1.22	0.71	1.10	**0.70**	0.86	0.85
I_3	**0.01**	0.09	0.05	0.06	0.06	0.05	0.05
I_4	0.26	0.21	**0.14**	0.21	**0.14**	**0.14**	**0.14**
I_5	0.63	0.58	**0.45**	0.54	0.48	0.50	0.48
I_6	0.92	0.71	0.57	0.70	**0.56**	0.63	0.58
I_7	**0.44**	0.75	0.53	0.62	0.52	0.57	0.53
I_8	0.82	0.67	0.59	0.64	0.59	**0.56**	0.58
I_9	1.70	1.61	**1.34**	1.63	1.42	1.51	1.44
I_{10}	1.55	1.31	1.05	1.29	**1.03**	1.10	**1.03**
I_{11}	**0.71**	1.74	1.09	1.59	1.05	1.34	1.02
I_{12}	3.53	2.07	1.54	2.00	**1.38**	1.71	1.64
Average gap_{lb}	1.11	1.01	0.74	0.95	**0.72**	0.82	0.76

Table 3
Average results for GRASP heuristics executing 200 iterations.

Id.	CPLEX	Average solution values					
		G	GPR1	GPR2	GPR1R2	GPR3	GPR4
I_1	**1.01**	1.75	1.26	1.37	1.13	1.32	1.28
I_2	1.77	1.94	1.32	1.49	**1.19**	1.47	1.39
I_3	**0.01**	0.35	0.27	0.21	0.18	0.29	0.23
I_4	0.26	0.28	0.20	0.25	**0.19**	0.21	0.21
I_5	0.63	0.84	0.66	0.71	**0.62**	0.71	0.67
I_6	0.92	0.92	0.73	0.81	**0.70**	0.79	0.75
I_7	**0.44**	1.08	0.89	0.89	0.79	0.93	0.87
I_8	0.82	0.83	0.71	0.75	**0.68**	0.74	0.72
I_9	1.70	1.88	1.60	1.74	**1.59**	1.71	1.70
I_{10}	1.55	1.54	1.26	1.41	**1.24**	1.35	1.32
I_{11}	**0.71**	2.07	2.42	2.33	2.03	2.62	2.31
I_{12}	3.53	3.16	2.24	2.61	**2.15**	2.53	2.45
Average gap_{lb}	1.11	1.47	1.13	1.21	**1.04**	1.22	1.16

Table 4
Average CPU time for GRASP heuristics executing 200 iterations.

Id.	Average CPU time (s)					
	G	GPR1	GPR2	GPR1R2	GPR3	GPR4
I_1	**57.05**	158.87	87.48	184.75	136.80	172.80
I_2	**151.01**	451.11	227.67	495.25	386.30	485.70
I_3	**2.48**	3.72	3.95	6.43	3.00	4.80
I_4	**750.30**	3718.32	1133.96	4132.28	3335.10	3735.10
I_5	**51.68**	121.30	75.65	142.81	106.00	158.90
I_6	**225.93**	720.47	331.15	822.49	631.60	860.40
I_7	**49.44**	99.47	75.97	119.72	86.30	132.80
I_8	**593.80**	1618.11	881.57	1837.78	1470.90	1870.30
I_9	**1762.68**	7592.92	2608.65	8749.56	7235.00	9435.00
I_{10}	**1765.84**	9421.46	2616.59	10 297.08	8190.90	12 390.00
I_{11}	**15.75**	33.95	25.07	42.37	29.90	49.30
I_{12}	**364.78**	1533.07	554.9	1652.09	1325.00	2201.80

Table 4 shows the average CPU time consumed by the GRASP heuristics. The first column shows the instance identifiers and the last six columns show the average CPU time spent by GRASP heuristics. We can observe that G and GPR2 obtained smaller average times. This was expected because G does not use path-relinking and GPR2 performs path-relinking just for one pair of solutions in each iteration, while GPR3, GPR4 and GPR1R2 perform path-relinking among all solutions of the elite set until the elite set is not updated. Although GPR1, GPR1R2, GPR3, and GPR4 spent average times larger than G and GPR2, they spent much less time than the CPU time limit of 25 000 s used for all CPLEX executions.

The results presented above show that GPR1, GPR1R2, GPR3, and GPR4 obtained better quality results, but required more computational time than G and GPR2. Thus, new experiments were executed to make a fair comparison of the performance of GRASP heuristics. The stopping criterion for the GRASP heuristics was changed to a time limit of 720 s. Ten executions of the GRASP heuristics were performed and the time limit to CPLEX was set to 7200 s. No optimal solution was found by CPLEX using this time limit, except for instance I_3, whose optimal solution was found in 442 s. Table 5 shows the results obtained for this new experiment. The first column shows the instance identifiers, followed by the gap_{lb} value obtained by CPLEX. The last six columns show the gap_{lb} values for the best results found by GRASP heuristics. The bold values in Table 5 indicate the best values. CPLEX, GPR1 and GPR4 obtained one each and GPR1R2 found six. The best average gap_{lb} value was obtained by GPR1R2. We observed that although GPR1, GPR1R2, GPR3, and GPR4 heuristics require more time per iteration (Table 4), the heuristics require fewer iterations to achieve good quality solutions.

Table 5
Best results for GRASP heuristics executing with a time limit.

Id.	CPLEX	Best solution values					
		G	GPR1	GPR2	GPR1R2	GPR3	GPR4
I_1	1.54	0.97	0.80	0.93	**0.76**	0.90	0.84
I_2	1.95	1.12	0.73	0.99	**0.66**	0.84	**0.66**
I_3	**0.01**	0.04	0.04	0.05	0.03	0.03	0.03
I_4	0.27	0.21	**0.19**	0.21	0.20	**0.19**	**0.19**
I_5	0.86	0.55	0.45	0.51	**0.43**	0.50	0.44
I_6	0.95	0.69	0.61	0.66	**0.58**	0.60	0.61
I_7	0.60	0.67	0.51	0.60	**0.49**	0.50	0.50
I_8	0.87	0.67	0.63	0.65	0.64	0.62	**0.61**
I_9	1.99	1.63	1.61	1.64	**1.60**	1.63	1.68
I_{10}	1.69	1.28	1.27	1.31	**1.27**	1.30	**1.27**
I_{11}	1.87	1.68	1.28	1.60	**1.19**	1.23	1.24
I_{12}	3.76	2.23	**1.80**	2.14	1.96	2.01	1.95
Average gap_{lb}	1.36	0.98	0.83	0.94	**0.82**	0.86	0.83

Table 6
Average results for GRASP heuristics executing with a time limit.

Id.	CPLEX	Average solution values					
		G	GPR1	GPR2	GPR1R2	GPR3	GPR4
I_1	1.54	1.78	**1.18**	1.21	**1.18**	1.72	1.65
I_2	1.95	1.93	1.24	1.36	**1.22**	1.67	1.52
I_3	**0.01**	0.35	0.22	0.14	0.18	0.35	0.33
I_4	0.27	0.28	**0.23**	0.25	0.24	0.24	0.24
I_5	0.86	0.85	0.64	**0.63**	**0.63**	0.84	0.78
I_6	0.95	0.92	**0.73**	0.78	**0.73**	0.80	0.76
I_7	**0.60**	1.09	0.82	0.78	0.78	1.08	1.04
I_8	0.87	0.82	**0.71**	0.75	0.72	0.73	0.72
I_9	1.99	1.88	**1.76**	1.79	**1.76**	1.80	1.81
I_{10}	1.69	1.54	1.44	1.47	**1.42**	1.45	1.46
I_{11}	**1.87**	3.30	2.37	2.01	2.30	3.38	3.31
I_{12}	3.76	3.32	**2.43**	2.70	2.54	2.66	2.63
Average gap_{lb}	1.36	1.51	1.15	1.16	**1.14**	1.39	1.35

Table 6 shows the average values obtained for gap_{lb} in 10 executions of the GRASP heuristics, using the time limit criterion of 720 s. The first column shows the instance identifiers, followed by the gap_{lb} value obtained by CPLEX. The last six columns show the average gap_{lb} values obtained by the GRASP heuristics. CPLEX, GPR1 and GPR1R2 obtained three best results and the best average gap_{lb} value was obtained by GPR1R2.

These results confirm that GPR1, GPR1R2, GPR3, and GPR4 heuristics performed better than G and GPR2 and that GPR1R2 presented the best performance. We can also observe the robustness of the proposed heuristics which obtain similar quality results both for best and average solution values.

Another experiment was performed using a subset of 27 small size instances of type 1, whose optimal values were found by CPLEX. The results presented in Table 7 show the computational results obtained by CPLEX and GPR1R2 heuristic, which presented the best performance in the previous experiments. The number of iterations of GPR1R2 was set to 200. The first column shows the instance identifiers, the second the cost of the optimal solution reached by CPLEX, and the third column the time demanded by CPLEX in seconds. The fourth column presents the gap_{op} value between the solution obtained by GPR1R2 and the *optimal* value and the fifth

Table 7
Comparison between CPLEX and GPR1R2 for small size instances, where CPLEX finds the optimum.

Instances	CPLEX	t_{CPLEX}(s)	GPR1R2(%)	t_{GPR1R2}(s)
5-eil51	437	12.31	**0.00**	1.00
10-eil51	440	74.38	**0.00**	1.00
15-eil51	437	2.04	**0.00**	1.00
5-berlin52	7991	201.80	**0.00**	1.20
10-berlin52	7896	89.17	**0.00**	1.10
15-berlin52	8049	75.93	**0.00**	1.10
5-st70	695	13 790.11	**0.00**	2.30
10-st70	691	4581.00	**0.00**	2.00
15-st70	692	883.50	**0.00**	2.00
5-eil76	559	83.70	0.36	2.70
10-eil76	561	254.30	0.53	2.40
15-eil76	565	49.66	0.35	2.50
5-pr76	108 590	99.29	**0.00**	2.70
10-pr76	109 538	238.13	**0.00**	2.20
15-pr76	110 678	261.94	0.15	2.30
10-rat99	1238	650.67	**0.00**	4.90
25-rat99	1269	351.15	0.63	4.70
50-rat99	1249	2797.58	0.72	4.90
25-kroA100	21 917	3513.57	**0.00**	4.70
50-kroA100	21 453	947.55	**0.00**	5.20
10-kroB100	22 440	4991.44	0.16	4.80
50-kroB100	22 355	2579.22	1.33	5.20
25-eil101	663	709.45	1.36	4.60
50-eil101	644	275.33	1.09	5.40
25-lin105	14 438	6224.55	**0.00**	5.10
50-lin105	14 379	1577.21	1.08	5.70
75-lin105	14 521	15 886.77	0.59	6.40
average values=	-	2266.73	0.25	3.30

column the average time spent by GPR1R2 in seconds. The gap_{op} value is calculated as follows:

$$gap_{op} = 100 * \left(\frac{vh - vcplex}{vcplex}\right), \quad (11)$$

where *vh* is the best *value* found by the GPR1R2 heuristic and *vcplex* the *value* of the optimal solution obtained by *CPLEX*.

The bold values in Table 7 indicate that GPR1R2 found the optimal solution. For the 27 instances, GPR1R2 obtained 15 optimal solutions and an average gap_{op} value equal to 0.25%. The average computational time is equal to 3.30 s, which is much less than the time spent by CPLEX. These results show the potential of GPR1R2 to find solutions of good quality in feasible computational time.

Another experiment was performed to verify the effectiveness of incorporating path-relinking to a GRASP heuristic by comparing the performance of the traditional GRASP (G) and GPR1R2. As we used some larger size instances, we set a longer CPU time limit equal to 1080 s as the termination criterion for both heuristics. Table 8 shows the best and average results obtained by G and GPR1R2 for these large size instances. The first column shows the instances name and the second their identifiers, followed by the number of vertices, number of clusters and the instance type. The sixth and seventh columns show the gap_h values calculated by Eq. (12) using the best results obtained by G and GPR1R2. The last columns show the gap_h values calculated using the average values obtained by G and GPR1R2, respectively.

The gap_h value is calculated as follows:

$$gap_h = 100 * \left(\frac{vh - v}{v}\right), \quad (12)$$

where *vh* is the value found by the heuristic and *v* is the best solution value found by G or GPR1R2.

We can observe that GPR1R2 was able to find all best results both for best and average solution values. As both heuristics consumed the same computational time, we confirm that introducing path-relinking to GRASP effectively improves the GRASP performance.

Table 8
Comparison between G and GPR1R2 heuristics for large size instances.

Instances	Id.	# vertices	# V_k	Type	Best results		Average results	
					G	GPR1R2	G	GPR1R2
49-pcb1173	I_{13}	1173	49	6	0.40	**0.00**	2.94	**0.00**
100-pcb1173	I_{14}	1173	100	6	0.54	**0.00**	2.87	**0.00**
144-pcb1173	I_{15}	1173	144	6	0.08	**0.00**	2.68	**0.00**
10-nrw1379	I_{16}	1379	10	6	0.66	**0.00**	2.33	**0.00**
12-nrw1379	I_{17}	1379	12	6	0.26	**0.00**	2.96	**0.00**
1500-10-503	I_{18}	1500	10	5	0.32	**0.00**	1.03	**0.00**
1500-20-504	I_{19}	1500	20	5	0.06	**0.00**	1.36	**0.00**
1500-50-505	I_{20}	1500	50	5	0.03	**0.00**	0.65	**0.00**
1500-100-506	I_{21}	1500	100	5	0.52	**0.00**	1.19	**0.00**
1500-150-507	I_{22}	1500	150	5	0.01	**0.00**	0.43	**0.00**
2000-10-a	I_{23}	2000	10	4	0.86	**0.00**	0.53	**0.00**
2000-10-h	I_{24}	2000	10	4	0.25	**0.00**	1.11	**0.00**
2000-10-z	I_{25}	2000	10	4	0.17	**0.00**	1.29	**0.00**
2000-10-x1	I_{26}	2000	10	4	0.91	**0.00**	0.49	**0.00**
2000-10-x2	I_{27}	2000	10	4	0.19	**0.00**	1.64	**0.00**
			Average gap_h=		0.35	**0.00**	1.57	**0.00**

Table 9
Target values for instances.

Id.	*difficult target*		*medium target*	
	Value	(%)	Value	(%)
I_{11}	63 600	1.21	63 700	1.38
I_4	916 400	0.15	916 550	0.17
I_{10}	130 280 000	0.31	130 340 000	0.36

4.3. Comparing GRASP heuristics using TTT plots

We executed another experiment to compare the performance of the traditional GRASP (G), and the GRASP with path-relinking strategies GPR4 and GPR1R2, based on Time-to-target (TTT) plots [33], which are used to analyze the behavior of randomized algorithms. A TTT plot is generated by executing an algorithm several times and measuring the time required to reach a solution at least as good as a target solution. In our experiments, each strategy was executed a hundred times. The *i*-th sorted running time t_i is associated with a probability $p_i = ((i-1/2)/100)$ and the points $z_i = (t_i, p_i)$, for $i = 1, \ldots, 100$ are plotted. Each plotted point shows the probability (vertical axis) for the strategy to achieve the target solution in the indicated time (horizontal axis).

We performed these experiments using three different instances: the instance I_{11}, whose best result was found by CPLEX using 25 000 s; the instance I_4, whose best result was obtained by GPR1R2 in average time of 4132.28 s; and the instance I_{10}, whose best result was obtained by GPR1R2 in average time of 10297.09 s.

Two target values were used for each instance: a medium and a difficult value. Table 9 shows the adopted target values and the percentage difference between the target value and the best value obtained in the previous experiments.

We also set a time limit for each experiment. If the target value was not found in this time limit, the execution was aborted. The time limit for instance I_{11} was 85 s and for instances I_4 and I_{10} was 3600 s.

Fig. 2(a) and (b) shows the results obtained for instance I_{11}. The probability of finding the medium target value in 16 s is 100%

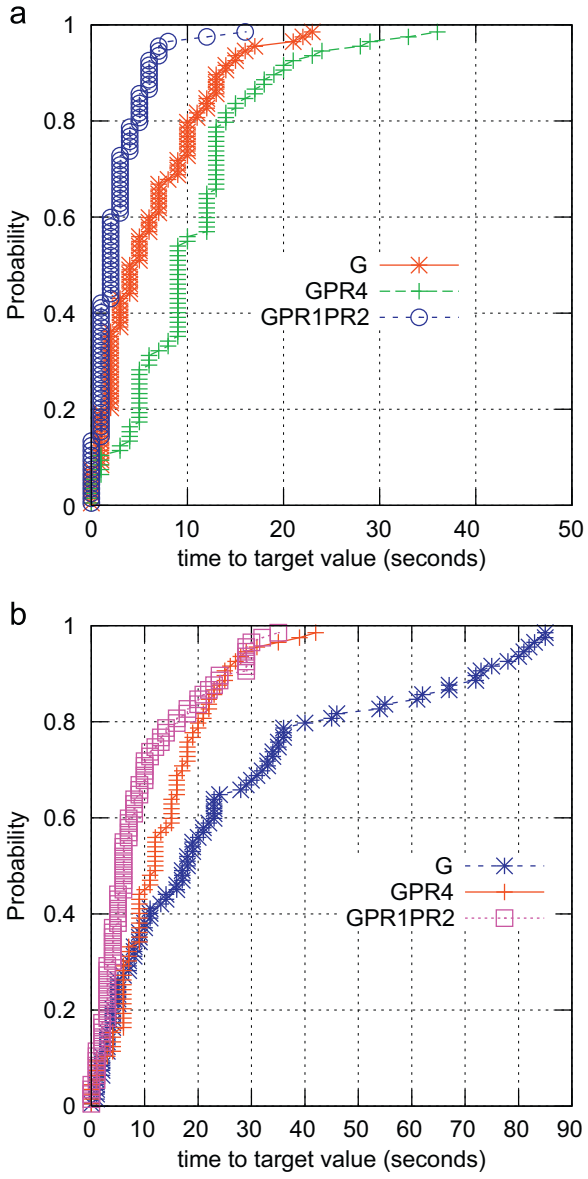

Fig. 2. Empirical distributions of time to target solution value for G, GPR4 and GPR1R2 for instance I_{11}: (a) medium target value; (b) difficult target value.

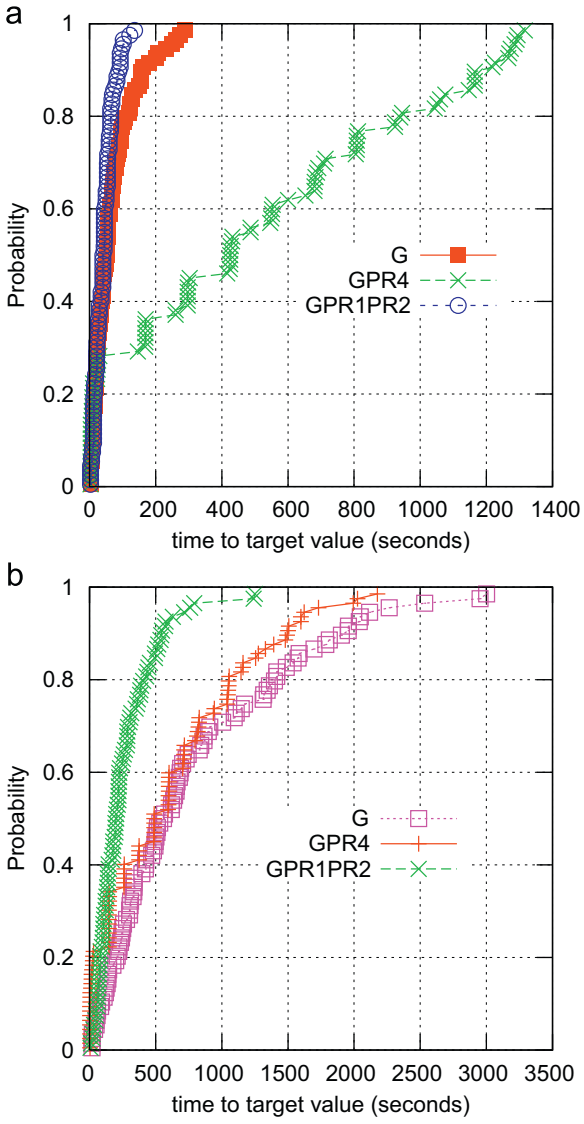

Fig. 3. Empirical distributions of time to target solution value for G, GPR4 and GPR1R2 for instance I_4: (a) medium target value; (b) difficult target value.

for GPR1R2, while for G and GPR4 is 95% and 85%, respectively. The results obtained for the difficult target show that, for GPR1R2 there is a probability of 100% to reach the target value in 35 s, while GPR4 needs 42 s to achieve the same probability. The G heuristic presents a probability equal to 75% to achieve the target in 35 s and in two executions, the execution time limit was reached.

Fig. 3(a) and (b) shows the results obtained for G, GPR4, and GPR1R2 for instance I_4. GPR1R2 presents 100% probability of finding the medium target value in 136 s, while G needs 288 s and GPR4 needs 1316 s. For the difficult target, GPR1R2 has 100% probability of finding the target value in 1250 s, while GPR4 needs 2177 s and G needs 3001 s.

Fig. 4(a) and (b) presents the results for instance I_{10}. GPR1R2 presents 100% probability of reaching the medium target value in 1056 s, while G needs 2692 s and GPR4 needs 2720 s. For the difficult target, GPR4 heuristic has 100 % probability of finding the target value in 3368 s, GPR1R2 presents 95% probability of finding the target in 3478 s and G has 46% probability of reaching the target in 3488 s. The GPR1R2 heuristic was not able to find the target value in the time limit in three executions, and G in 53 executions.

For all instances, we can observe that, for both targets, GPR1R2 has higher probability than G and GPR4 to find a target solution in less computational time. For medium values, G presented higher probabilities than GPR4 and for difficult values, GPR4 presented higher probabilities than G.

5. Comparison between the proposed heuristics and a Genetic Algorithm

The proposed heuristics were compared with a Genetic Algorithm (GA) [17] which solves the CTSP without a prespecified order of visiting the clusters. The GA was implemented in this work, because it was not possible to access the original code and every recommendation made by the authors in [34] were followed.

The developed *Two-Level Genetic Algorithm* (TLGA) [17] constructs solutions for the CTSP at two levels: one called lower level and other called higher level. At the lower level, a GA tries to find

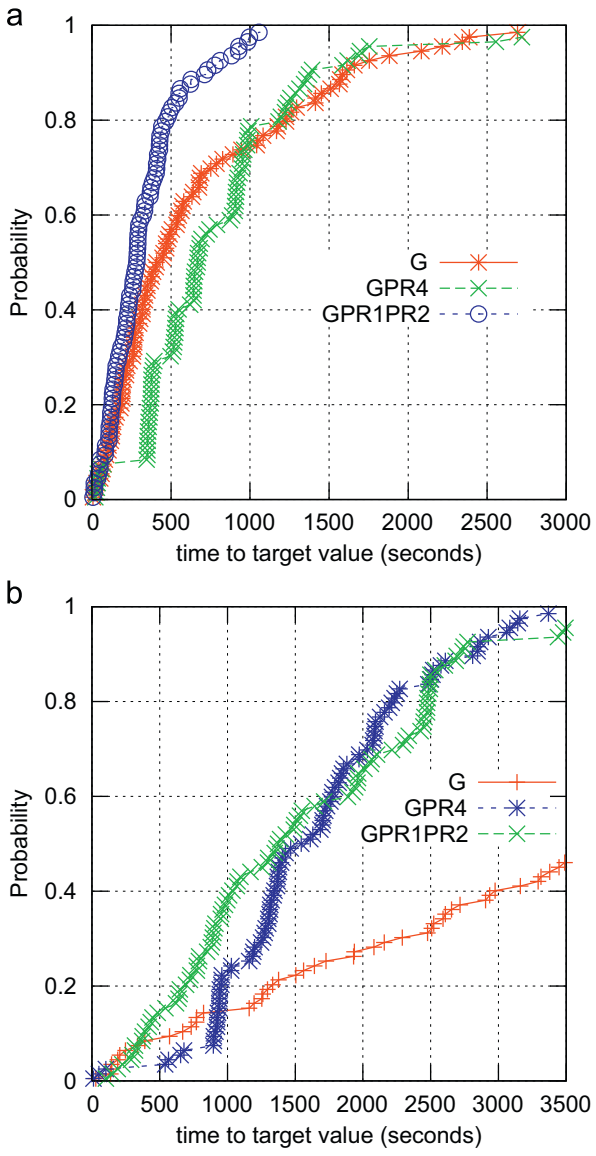

Fig. 4. Empirical distributions of time to target solution value for G, GPR4 and GPR1R2 for instance I_{10}: (a) medium target value; (b) difficult target value.

the shortest Hamiltonian tour for each cluster and returns tours $T_k, k=1,\ldots,m$ for each cluster $V_1, V_1, \ldots, V_m$.

At the higher level, TLGA tries to find a shortest tour for the CTSP based on the tours generated by the lower level algorithm. One crossover and two mutation operators are applied to produce the next generation.

The TLGA algorithm stops when the best solution is not updated for a *maxger* number of consecutive generations.

Algorithm 8 shows the TLGA procedure, adapted from the TLGA flowchart found in [17]. The procedure *Lower_Level* generates the initial T_k tours. Then, the *procedure Initialize_Population* creates an initial population of chromosomes. The initial population is created by selecting two adjacent vertices in each tour T_i as end vertices for cluster V_i and specifying which one will be the start vertex and which one will be the end vertex. Then a sequence for visiting all the clusters is randomly defined and a Hamiltonian cycle C to the CTSP is found using the defined start and end vertices and the visiting sequence.

The crossover operator and the two mutation operators are applied according to the probabilities CF, MF_1, MF_{21} and MF_{22}, respectively.

The crossover and the first mutation operators are used to change the visiting sequence of clusters. The second mutation operator is used to change the visiting sequence of vertices inside a cluster without changing the visiting order of clusters. More details about these operators can be obtained in [17].

In line 7, a pair of parents (S_{p_i}, S_{p_j}), $i \neq j$ is selected from the current population and the crossover operation `Crossover`(S_{p_i}, S_{p_j}) is performed in line 9 to create a new offspring S_{AG}. Then, the first mutation operation `Cluster_Mutation`(S_{p_i}, S_{p_j}) is performed in line 12 and the second mutation procedure `Gene_Mutation`(k) in line 17.

The population P is updated with the solution S_{AG} in line 21. If the number of generations without improving the solution is greater than *maxger*, the algorithm terminates and returns the best solution S^*.

Algorithm 8. Genetic Algorithm (TLGA), adapted from [17]

```
1:  Lower_Level;
2:  Initialize_Population;
3:  S* ← best_cost_solution;
4:  impiter ← 1;
5:  while (impiter) ≤ maxger do
6:    Evaluate the population P={p1, p2, …, ph} by the fitness
      function;
7:    Select parents (Spi, Spj) in P for next generation;
8:    if random(1,0) < CF then
9:      SAG ← Crossover(Spi, Spj)
10:   end if
11:   if random(1,0) < MF1 then
12:     SAG ← Cluster_Mutation(Spi, Spj);
13:   end if
14:   if random (1,0) < MF21 then
15:     for k=1 to numclusters do
16:       if random(1,0) < MF22 then
17:         SAG ← Genes_Mutation(k);
18:       end if
19:     end for
20:   end if
21:   Update Population P with the solution SAG;
22:   if f(SAG) < f(S*) then
23:     S* ← SAG;
24:     impiter ← 0;
25:   else
26:     impiter ← impiter+1;
27:   end if
28: end while
29: return S*;
```

We implemented the TLGA algorithm using the characteristics and parameters available in [34]. The TLGA and proposed heuristics were implemented in the C programming language and executed on the same computer described in Section 4.

The instances used in [34] were not available, so we compared the heuristics using the instances already described in the previous section.

The parameters used for TLGA were taken from [34]. The population size is proportional to the number of vertices n in each instance and the population size l for the lower level algorithm is different from the population size h for the higher level algorithm. The crossover fraction CF was set to 0.76 and the mutation fraction MF_1 was set to 0.06. The mutation fractions MF_{21} and MF_{22} were set to 0.13 and 0.50, respectively. The maximum

Table 10
Comparison between GPR1R2 and TLGA for small size instances.

Instances	TLGA (%)	t_{TLGA}(s)	GPR1R2(%)	t_{GPR1R2}(s)
5-eil51	8.47	**0.40**	**0.00**	1.00
10-eil51	2.73	**0.40**	**0.00**	1.00
15-eil51	7.78	**0.40**	**0.00**	1.00
5-berlin52	1.44	**0.60**	**0.00**	1.00
10-berlin52	10.18	**0.40**	**0.00**	1.00
15-berlin52	14.69	**0.40**	**0.00**	1.00
5-st70	0.86	**1.60**	**0.00**	2.20
10-st70	1.88	**1.00**	**0.00**	1.80
15-st70	7.23	**0.80**	**0.00**	1.80
5-eil76	3.94	**1.80**	**0.54**	2.40
10-eil76	9.45	**1.20**	**0.71**	2.40
15-eil76	2.48	**1.20**	**0.35**	2.40
5-pr76	1.78	**2.00**	**0.92**	2.60
10-pr76	1.02	**1.20**	**0.01**	2.40
15-pr76	5.95	**1.20**	**0.15**	2.40
10-rat99	6.70	**3.40**	**0.24**	4.60
25-rat99	23.48	**2.40**	**1.02**	4.60
50-rat99	37.15	**2.40**	**2.24**	4.60
25-kroA100	5.69	**2.20**	**0.00**	4.80
50-kroA100	22.23	**2.80**	**1.02**	5.00
10-kroB100	2.49	**3.80**	**0.07**	4.80
50-kroB100	25.36	**2.20**	**0.16**	5.00
25-eil101	6.33	**2.20**	**1.51**	4.80
50-eil101	20.34	**2.20**	**2.95**	5.00
25-lin105	19.39	**2.00**	**0.15**	5.20
50-lin105	26.29	**3.20**	**0.54**	5.80
75-lin105	56.62	**4.60**	**0.89**	6.20
Average values	12.29	**1.80**	**0.44**	3.21

number of generation *maxger* was also proportional to the number of vertices n in each instance.

The first experiment was executed to compare TLGA and GPR1R2 performance using small size instances of type 1, where $51 \leq n \leq 105$. The stopping criterion for GPR1R2 was the maximum number of iterations and was set to 200. For TLGA, the number of iterations is limited to the maximum number of generations (*maxger*) without updating the best solution. For this experiment the value of *maxger* was set to 10. The size of the lower level population l was set to 5 and the size of higher population h was set to 10. The number of executions was set to 5 for both GPR1R2 and TLGA.

The obtained results are shown in Table 10. The first column shows the instance identifier. The second column shows the gap_{op} value obtained by TLGA algorithm. The third column presents the computational time t_{TLGA} demanded by TLGA in seconds. The fourth column shows the gap_{op} value obtained by GPR1R2 and the fifth column the average time spent by GPR1R2 in seconds. The bold values in Table 10 indicate the best values. GPR1R2 and TLGA obtained an average gap_{op} value equal to 0.44% and 12.29%, respectively. The average computational time of GPR1R2 was 3.21 s and of TLGA was equal to 1.80 s.

The results presented in Table 10 show that GPR1R2 obtained better quality results, but required more computational time than TLGA.

A new experiment was executed to make a new comparison between the performance of G, GPR1R2 and TLGA. For this new experiment, the instances previously used in Section 4 were used. The stopping criterion for the algorithm was changed to a time limit of 720 s, except for instance I_3, whose optimal solution was found in 442 s. Ten executions were performed for each instance for all algorithms.

The parameters for TLGA algorithm were set as follows. If $n \leq 105$, then the value h was set to 10, if $(105 < n \leq 442)$, h was set to 80, and if $(442 < n \leq 783)$, h was set to 90. The size of the population l for the lower level algorithm, crossover and mutation fractions, and other parameters are the same presented in [34].

Table 11
Comparison between GRASP heuristics and TLGA for the best results.

Id.	Best results		
	TLGA (%)	G(%)	GPR1R2(%)
I_1	1.08	0.97	**0.76**
I_2	9.07	1.12	**0.66**
I_3	0.05	0.04	**0.03**
I_4	2.70	0.21	**0.20**
I_5	0.56	0.55	**0.43**
I_6	4.39	0.69	**0.58**
I_7	0.81	0.67	**0.49**
I_8	6.99	0.67	**0.64**
I_9	26.32	1.63	**1.60**
I_{10}	23.61	1.28	**1.27**
I_{11}	1.72	1.68	**1.19**
I_{12}	33.71	2.23	**1.96**
Average gap_{lb}	9.25	0.98	**0.82**

Table 12
Comparison between GRASP heuristics and TLGA for average results.

Id.	Average results		
	TLGA (%)	G(%)	GPR1R2(%)
I_1	6.27	1.78	**1.18**
I_2	16.86	1.93	**1.22**
I_3	0.33	0.35	**0.18**
I_4	3.28	0.28	**0.24**
I_5	1.84	0.85	**0.63**
I_6	6.08	0.92	**0.73**
I_7	4.72	1.09	**0.78**
I_8	8.22	0.82	**0.72**
I_9	27.26	1.88	**1.76**
I_{10}	24.72	1.54	**1.42**
I_{11}	4.34	3.30	**2.30**
I_{12}	38.66	3.32	**2.54**
Average gap_{lb}	11.88	1.51	**1.14**

Table 11 shows the best results obtained for this new experiment. The first column shows the instance identifiers, followed by the gap_{lb} values obtained by the algorithms. The bold values indicate the best values.

We can observe that G and GPR1R2 found better solutions than TLGA for all instances. G obtained an average gap_{lb} value equal to 0.98%, GPR1R2 a value equal to 0.82%, and TLGA a value equal to 9.25%.

Table 12 shows the average results obtained for this new experiment. The bold values indicate the best values. TLGA obtained an average gap_{lb} value equal to 11.88%, while G got a value equal to 1.51% and GPR1R2 obtained a value equal to 1.14%.

The results presented in Tables 11 and 12 show that G and GPR1R2 were able to find solutions of better quality than TLGA using the same computational time. So, the use of GRASP and path-relinking to solve the CTSP without specifying the order of visiting the clusters allowed to achieve better results than a Genetic Algorithm developed to solve the same problem.

6. Conclusions

In this paper, we proposed six heuristics for the Symmetric Euclidean Clustered Traveling Salesman Problem (SECTSP) based on the GRASP heuristic. Several path-relinking strategies incorporated to GRASP were investigated. The comparison of the results obtained by the proposed heuristics and by the mathematical

formulation implemented with the CPLEX software showed that the proposed heuristics were able to produce good quality solutions for the SECTSP in reasonable computational time. The computational results showed that the heuristic GPR1R2 which uses path-relinking in each iteration and as a post-optimization strategy outperformed other heuristics which incorporate other different path-relinking strategies applied to the traditional GRASP (G).

The proposed heuristics G and GPR1PR2 were also compared to a Genetic Algorithm developed for solving the CTSP without specifying the order of visiting the clusters. Computational results showed that the proposed heuristics were able to find better quality solutions in the same computational time.

The computational results obtained in this work were applied for symmetric Euclidean instances, but the heuristics that were presented can be easily adapted to produce solutions for another type of instances.

Furthermore, it is important to observe that the proposed heuristics obtained good quality solutions for a generic version of CTSP, where no order of visiting the clusters is specified a priori. The CTSP is usually simplified in the literature by specifying an order for visiting the clusters.

Acknowledgments

Thanks are due to Jean-Yves Potvin, *Département d'informatique et de Recherche Opérationnelle, Université de Montréal*, for supporting the bibliographic material and David S. Johnson, AT&T Labs Research, who provided the codes to generate instances of type 2. The first author would also like to thank the IFES "*Instituto Federal de Educação, Ciência e Teconologia do Espírito Santo*" for their supporting this work. This work was partially supported by Brazilian Research Agencies: CNPq "*Conselho Nacional de Desenvolvimento Científico e Tecnológico*", CAPES "*Coordenação de Aperfeiçoamento de Pessoal de Nível Superior*", and FAPERJ "*Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro*".

References

[1] Chisman JA. The clustered traveling salesman problem. Computers and Operations Research 1975;2(2):115–9.
[2] Weintraub A, Aboud J, Fernandez C, Laporte G, Ramirez E. An emergency vehicle dispatching system for an electric utility in Chile. Journal of the Operations Research Society 1999;50:690–6.
[3] Lokin FCJ. Procedures for travelling salesman problems with additional constraints. European Journal of Operational Research 1979;3(2):135–41.
[4] Laporte G, Palekar U. Some applications of the clustered travelling salesman problem. Journal of the Operations Research Society 2002;53(9):972–6.
[5] Ghaziri H, Osman IH. A neural network for the traveling salesman problem with backhauls. Computers & Industrial Engineering 2003;44(2):267–81.
[6] Anily S, Bramel J, Hertz A. A 5/3-approximation algorithm for the clustered traveling salesman tour and path problems. Operations Research Letters 1999;24(1–2):29–35.
[7] Gendreau M, Laporte G, Potvin JY. Heuristics for the clustered traveling salesman problem. Technical Report CRT-94-54, Centre de recherche sur les transports, Universidade de Montreal, Montreal, Canada; 1994.
[8] Laporte G, Potvin J-Y, Quilleret F. A tabu search heuristic using genetic diversification for the clustered traveling salesman problem. Journal of Heuristics 1996;2(3):187–200.
[9] Potvin J-Y, Guertin F. A genetic algorithm for the clustered traveling salesman problem with an a priori order on the clusters. Technical Report CRT-95-06, Centre de recherche sur les transports, Université de Montréal, Canada; 1995.
[10] Arkin EM, Hassin R, Klein L. Restricted delivery problems on a network. Networks 1997;29(4):205–16.
[11] Gendreau M, Laporte G, Hertz A. An approximation algorithm for the traveling salesman problem with backhauls. Operations Research 1997;45(4):639–41.
[12] Guttmann-Beck N, Hassin R, Khuller S, Raghavachari B. Approximation algorithms with bounded performance guarantees for the clustered traveling salesman problem. Algorithmica 2000;28(4):422–37.
[13] Christofides N. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University; 1976.
[14] Hoogeveen JA. Analysis of Christofides' heuristic: some paths are more difficult than cycles. Operations Research Letters 1991;10(5):291–5.
[15] Jongens K, Volgenant T. The symmetric clustered traveling salesman problem. European Journal of Operational Research 1985;19(1):68–75.
[16] Potvin J-Y, Guertin F. The clustered traveling salesman problem: a genetic approachIn: Osman IH, Kelly J, editors. Meta-heuristics: Theory & applications. Norwell, MA, USA: Kluwer Academic Publishers; 1996. p. 619–31.
[17] Ding C, Cheng Y, He M. Two-level genetic algorithm for clustered traveling salesman problem with application in large-scale TSPs. Tsinghua Science and Technology 2007;12(4):459–65.
[18] Festa P, Resende M. An annotated bibliography of GRASP, part I: algorithms. International Transactions in Operational Research 2009;16:1–24.
[19] Festa P, Resende M. An annotated bibliography of GRASP, part II: applications. International Transactions in Operational Research 2009;16:131–72.
[20] Resende M, Ribeiro C. Greedy randomized adaptive search procedures. In: Glover F, Kochenberger G, editors. Handbook of metaheuristics. Kluwer Academic Publishers; 2002. p. 219–49.
[21] Martí R, Laguna M, Glover F. Principles of scatter search. European Journal of Operational Research 2006;169(2):359–72.
[22] Resende MG, Martí R, Gallego M, Duarte A. GRASP and path relinking for the max–min diversity problem. Computers and Operations Research 2010;37(3):498–508.
[23] Reinelt G. TSPLIB. Universitt Heidelberg, Institut für Informatik, Heidelberg, Alemanha, 2007, accessed 17 September 2007. URL 〈http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95〉.
[24] Miller CE, Tucker AW, Zemlin RA. Integer programming formulation of traveling salesman problems. Journal of the ACM 1960;7(4):326–9.
[25] Feo T, Resende M. Greedy randomized adaptive search procedures. Journal of Global Optimization 1995;6(2):109–33.
[26] Johnson DS, McGeoch LA. The traveling salesman problem: a case study in local optimizationIn: Aarts EHL, Lenstra JK, editors. Local search in combinatorial optimization Londres. John Wiley and Sons; 1997. p. 215–310.
[27] Prais M, Ribeiro C. Reactive GRASP: an application to a matrix decomposition problem in TDMA traffic assignment. INFORMS Journal on Computing 2000;12:164–76.
[28] Laguna M, Martí R. GRASP and path relinking for 2-layer straight line crossing minimization. INFORMS Journal on Computing 1999;11(1):44–52.
[29] Resende M, Ribeiro C. GRASP with path-relinking: recent advances and applicationsIn: Ibaraki T, Nonobe K, Yagiura M, editors. Metaheuristics: Progress as real problem solvers. Springer Verlag; 2005. p. 29–63.
[30] Johnson DS, McGeoch LA. Experimental analysis of heuristics for the STSP. In: Gutin G, Punnen A, editors. The traveling salesman problem and its variations Dordrecht, Holanda: Kluwer Academic Publishers; 2002. p. 369–443.
[31] Applegate D, Bixby R, Chvátal V, Cook W. Concorde TSP solver. William Cook, School of Industrial and Systems Engineering, Georgia Tech., 2007, accessed 22 December 2007. 〈http://www.tsp.gatech.edu/concorde/index.html〉.
[32] CPLEX, ILOG CPLEX 11.2 User's manual and reference manual, 2009, ILOG S.A., 2009, accessed 06 January 2009. 〈http://www.ilog.com〉.
[33] Aiex R, Resende M, Ribeiro C. TTT plots: a perl program to create time-to-target plots. Optics Letters 2007;1(4):355–66.
[34] Cheng Y, Ding C. Private communication.