Iranian Journal of Operations Research
Vol. 6, No. 1, 2015, pp. 65-78

A Metaheuristic Algorithm for the Minimum Routing Cost Spanning Tree Problem

S. Sattari[1], F. Didehvar[2,*]

The routing cost of a spanning tree in a weighted and connected graph is defined as the total length of paths between all pairs of vertices. The objective of the minimum routing cost spanning tree problem is to find a spanning tree such that its routing cost is minimum. This is an NP-Hard problem that we present a GRASP with path-relinking metaheuristic algorithm for it. GRASP is a multi-start algorithm that in each iteration constructs a randomized greedy solution and applies local search to it. Path-relinking stores elite solutions and to find better solutions explores the paths between different solutions. Experimental results show the performance of our algorithm on many benchmark problems compared to the other algorithms.

Keywords: *Graph, Spanning Tree, Routing Cost, Metaheuristic*.

Manuscript was received on --/--/----, revised on --/--/---- and accepted for publication on --/--/----.

1. Introduction

Consider an undirected connected graph $G(V, E)$ that has non-negative edge weights. For any spanning tree T of G, we show the path length between any two vertices v and u by $d_T(u, v)$ and call it routing cost of these vertices. The routing cost of the spanning tree T is defined as sum of these costs i.e. $C(T) = \sum_{u,v\in V} d_T(u, v)$. The minimum routing cost spanning tree (MRCST) problem seeks a spanning tree T such that $C(T)$ is the minimum among all spanning trees of the graph.

This optimization problem was introduced by Hu [8] and later Johnson et al. [9] proved that MRCST is an NP-Hard problem. Applications of this problem are in the bridging of heterogeneous networks [3], and multiple sequence alignment problem of computational biology [6].

One of the approaches for solving optimization problems is using a metaheuristic approach. Greedy randomized adaptive search (GRASP) is a multi-start algorithm, which in each iteration constructs a randomized greedy solution and then applies a local search to the obtained solution. Path-relinking is a method that is used with other algorithms to improve their results. It stores elite solutions and to find better solutions, explores the trajectories that connect different solutions. In this paper, we present a GRASP with path-relinking metaheuristic algorithm for the MRCST problem and report its performance on a set of benchmark instances.

Throughout this paper, we use symbol n for the number of vertices of graph i.e. $|V|$, so every solution to MRCST problem consists of $n - 1$ edges. In the rest of the paper, first we review related

[*]Corresponding Author.
[1]Department of Mathematics and Computer Science, Amirkabir University of Technology, Tehran, Iran Email: s.sattari@aut.ac.ir.
[2]Department of Mathematics and Computer Science, Amirkabir University of Technology, Tehran, Iran Email: didehvar@aut.ac.ir.

works. Then, in section 3, we give an introduction to GRASP and path-relinking algorithms. In Section 4, we present our algorithm for MRCST problem. Section 5 reports experimental results, and the last section concludes the paper.

2. Related Works

The MRCST problem is a special case of optimal communication spanning tree (OCST) problem [8]. In OCST problem each pair of vertices has a demand that is multiplied to their path length for computing their communication cost. In this section we generally consider works on MRCST problem.

The MRCST problem has been studied from different approaches. In the field of approximation algorithms, Wong [19] presented a 2-approximation algorithm. In his algorithm, shortest paths trees are constructed from each vertex as root. The output of the algorithm is a tree, which sum of routing costs from its root is the least.

Wu et al. [21] provided a polynomial time approximation scheme. In another work, Wu et al [20] presented approximation algorithms with factors 2, 15/8, and 3/2, respectively with time complexities $O(n^2)$, $O(n^3)$, and $O(n^4)$.

Ahuja and Murty [1] provided a branch and bound algorithm and a heuristic algorithm for the OCST problem. This heuristic method at first constructs a tree greedily, and then applies a local search procedure. In this local search algorithm, every edge of the tree is tested for deletion, and alternative edges are inserted into the tree. Algorithm selects the best pair that yields the best possible spanning tree.

Fischetti et al. [6] presented a branch & price algorithm for MRCST problem. In generation of initial solutions they used a local search approach to improve the initial solution. This local search approach, tries inserting edges into the current tree and breaking the resulting cycle by deleting another edge. It selects the best combination that results in the lowest cost possible tree.

Campos and Ricardo [3] proposed a heuristic algorithm. This algorithm constructs a spanning tree by adding one vertex at each time to the tree. It uses many factors like degree of vertices, and weights of their edges to choose the next vertex. The algorithm has some parameters that they found them by simulation.

Julstrom gave two genetic algorithms and a stochastic hill-climber [10]. These genetic algorithms differ in the representation method of spanning trees. One uses edge sets and the other uses blob code [12]. The experimental results show superiority of hill-climber to both genetic algorithms. The hill-climber starts from a random spanning tree and in each step tries modifying this tree randomly. If this new tree has lower cost, it becomes current tree. This algorithm runs for 10000n iterations.

Singh [17] proposed a perturbation based local search algorithm. This algorithm, constructs a solution using a Prim [13] like algorithm, and tries to improve it. Constructing initial solution starts with selecting a random vertex. Next, in each step either the least cost possible edge, or a random edge is added to the current tree. Probability of this decision is constant, and probability of selecting a random edge is inversely related to its weight. The algorithm iteratively applies a local search to the current solution, and if it finds a better solution than current solution, it becomes the current solution. The local search method removes a random edge and inserts best possible alternative edge into the tree. Deleting this new edge is forbidden in the next iteration. After some non-improving iterations, algorithm modifies the current solution by deleting and inserting some random edges. They tested their algorithm on the dataset introduced by [10] and found better results.

Singh and Sundar [18] used an artificial bee colony (ABC) approach with local search. Their method constructs initial solutions by an algorithm like Prim's [13]. This algorithm starts by choosing a random vertex for the tree. Then in each step, one random edge is added to the tree. At the beginning, with a constant probability, algorithm decides probability of adding edges to be inversely related to their weight or square of their weight for all of the edges. Artificial bee colony algorithm is a swarm algorithm that imitates honey bee's behavior. To improve solutions of this method a local search like [1] is applied to the results of ABC algorithm. They showed superiority of this method, by running it on the same dataset as [10, 17].

3. GRASP and Path-relinking

Greedy randomized adaptive search procedure (GRASP) is a multi-start metaheuristic method. In each iteration it has two phases. In the first phase, it constructs a solution using a randomized greedy algorithm and in the second phase, it runs a local search on the obtained solution. The best solution over all iterations is the output of the algorithm. Stopping criteria for this algorithm can be for example, a maximum number of iterations. This method was proposed by Feo [5] for the set cover problem and it has been used in many other problems [15]. The outline of a standard GRASP algorithm for a minimization problem is as follows.

```
**Algorithm 1**: GRASP
**Output**: solution $s_*$
1. $s_* \leftarrow \infty$
2. **while** stopping criteria is not met **do**
3.    $s_1 \leftarrow$ GreedyRandomizedConstruction
4.    $s \leftarrow$ LocalSearch($s_1$)
5.    **if** $s < s_*$ **then** $s_* \leftarrow s$
6. **end**
7. **return** $s_*$
```

In the standard GRASP method, construction phase builds a solution by adding elements one at a time. In each step a list of elements that can be added to current solution is generated. These elements are ordered according to a greedy function and its best elements are inserted into a restricted candidate list (RCL). Then a random member of RCL is selected that is added to the current partial solution. After updating current solution this process is continued until there is no element that can be added to the solution. There are different methods of constructing a restricted candidate list, such as limiting the number of its elements or considering quality of the elements. Some alternative construction methods were also proposed [15].

Incorporating memory usually can enhance the basic GRASP method. Path-relinking is one of such methods that can be used with GRASP to improve quality of its solutions, and to increase speed of finding better solutions. Path-relinking is a procedure that explores trajectories connecting different solutions in order to find other solutions. This method was proposed by Glover [7] in conjunction with scatter search, but it can be used with other methods. The input for this algorithm consists of two solutions. It considers a path of solutions starting from one of the inputs and ending in the other. In each step, in order to reach the target solution, algorithm changes current solution slightly. The best solution found in this path, is retuned by the algorithm as its output. Algorithm 2 shows general path-relinking method for a minimization problem.

```
Algorithm 2: Path-relinking ($x_s, x_t$)
Input: solutions $x_s, x_t$
Output: solution $s_*$
1. $s_* \leftarrow Min(x_s, x_t)$
2. $s \leftarrow x_s$
3. while $s \neq x_t$ do
4.    Find best move $m$ such that $Distance(s \oplus m, x_t) < Distance(s, x_t)$
5.    $s \leftarrow s \oplus m$
6.    if $s < s_*$ then $s_* \leftarrow s$
7. end
8. return $s_*$
```

To use path-relinking with other methods, usually good solutions are stored in a pool. There are two general methods of using path-relinking with GRASP. In the first method, after each local search phase of algorithm, path-relinking is executed once. In the second method during execution of GRASP or at the end of it, path-relinking is executed between all elite solutions.

The first approach usually yields better results. In this approach, after finding a solution by GRASP, one elite solution from the pool is selected randomly, and path-relinking procedure is executed for these two solutions. The obtained solution is tested for insertion into the pool. If the pool is not full, it is inserted; otherwise the new solution may replace one of its members. There are different replacement strategies. In one of the approaches, if the new solution is better than the worst member of the pool, the new solution replaces it. In another approach, all members of pool that are worse than the new solution are listed, and it replaces the one which is more similar to it. In algorithm 3 we show steps of the GRASP with path-relinking algorithm.

```
Algorithm 3: GRASP-Path-relinking
Output: solution $s_*$
 1. $s_* \leftarrow \infty$
 2. $P \leftarrow \emptyset$
 3. while stopping criteria is not met do
 4.    $s_1 \leftarrow$ A greedy random solution
 5.    $s_2 \leftarrow$ LocalSearch($s_1$)
 6.    if $P \neq \emptyset$ then
 7.       $t \leftarrow$ A random member of $P$
 8.       Set $x_s \leftarrow s_2, x_t \leftarrow t$ or $x_s \leftarrow t, x_t \leftarrow s_2$
 9.       $s \leftarrow$ Path-relinking($x_s, x_t$)
10.    end
11.    Update $P$ with $s$
12.    if $s < s_*$ then $s_* \leftarrow s$
13. end
14. return $s_*$
```

GRASP with path-relinking was introduced by Laguna and Marti [11]. A detailed description of this method along with its extensions and applications has been given by Resende and Ribeiro [14].

4. GRASP with Path-relinking for MRCST Problem

In this section, we present our GRASP with path-relinking algorithm for the minimum routing cost spanning tree problem. First, we describe our GRASP algorithm. Then, we present a path-relinking algorithm, and finally we give our GRASP with path-relinking algorithm for the MRCST problem.

4.1 GRASP Algorithm

Every GRASP algorithm needs two components: a greedy randomized construction algorithm and a local search algorithm. In the basic GRASP after constructing RCL, one element is chosen randomly. Bresina [2] proposed a construction method which uses a rank function for all candidate elements and by incorporating a bias function applies a different probability distribution for random selection. The benefit of this method is that by using a problem related rank function more effective elements can be selected. In our construction algorithm, we follow a similar approach.

To construct a solution at first using Dijkstra's algorithm [4] we generate all shortest paths trees, and we calculate routing costs of them and memorize roots of the two least cost trees. To generate a spanning tree, we randomly select one of these roots, call it r, and then in a way similar to the Prim's algorithm [13] we add other vertices to the tree. In each step, for each edge (v, w) that vertex v is inside tree and vertex w is outside of it, the probability of selecting this edge is inversely related to its weight multiplied to the distance of w to vertex r. The steps of this algorithm are given in algorithm 4.

Algorithm 4: RandomizedGreedy-MRCST
Output: tree T

```
Initialization:
 1. for $i = 1$ to $n$
 2.   $t \leftarrow$ Shortest paths tree with root $v_i$
 3.   $cost[i] \leftarrow C(t)$
 4. end
 5. Find $r_1$ such that $\forall i \in \{1, \dots, n\} \cdot cost[r_1] \leq cost[i]$
 6. Find $r_2 \neq r_1$ such that $\forall i \in \{1, \dots, n\} - \{r_1\} \cdot cost[r_2] \leq cost[i]$

Construction:
 1. Randomly set $r = r_1$ or $r = r_2$
 2. Put $r$ in $T$
 3. for $i = 1$ to $n - 1$ do
 4.   if $v \in T \wedge w \in V - T$ then
 5.     $P(v, w) = \frac{c(v,w).d_T(r,w)}{\sum_{v \in T, w \in V-T} c(v,w).d_T(r,w)}$
 6.   else
 7.     $P(v, w) = 0$
 8.   end
 9.   Select an edge $(v, w)$ with probability $P(v, w)$
10.   Put vertex $w$ and edge $(v, w)$ in $T$
11. end
12. return $T$
```

In local search part of GRASP, we use the method of Ahuja and Murty [1]. The following algorithm shows details of this best improvement local search method.

Algorithm 5: LocalSearch-MRCST (T)
Input: tree T
Output: modified tree T

```
 1. $b \leftarrow 1$
 2. **while** $b = 1$ **do**
 3.   $c_{min} \leftarrow C(T)$
 4.   $b \leftarrow 0$
 5.   **foreach** edge $e \in T$ **do**
 6.     $T = T - \{e\}$
 7.     Find set $F$ of edges different from $e$ such that each of them joins components of $T$
 8.     **foreach** edge $f \in F$ **do**
 9.       **if** $C(T \cup \{f\}) < c_{min}$ **then** $c_{min} \leftarrow C(T \cup \{f\}), e_{del} \leftarrow e, e_{ins} \leftarrow f, b \leftarrow 1$
10.     **end**
11.     $T = T \cup \{e\}$
12.   **end**
13.   **if** $b = 1$ **then** $T = T - \{e_{del}\} \cup \{e_{ins}\}$
14. **end**
15. **return** $T$
```

In each iteration of this algorithm, an edge is removed from current tree, and another edge is added to it such that it remains a tree. It tries deleting each edge of the tree, and then finds all possible replacement edges and calculates resulting routing costs. If no replacement produces a tree with lower cost than current tree, the local search ends, otherwise the best pair of edges is chosen and this change is applied to the current tree and algorithm continues with this new spanning tree. By using this local search algorithm and construction algorithm 4, we can have a GRASP algorithm.

4.2 Path-relinking Algorithm

Our path-relinking algorithm has as input a starting spanning tree T_{start} and a target spanning tree T_{end}. Consider A as the set of T_{start} edges not present in T_{end}, and B as the set of T_{end} edges not present in T_{start}. In each step, we remove an edge of the current tree which belongs to A; this results in two components. We add another edge such that it connects these components; we choose this edge from the set B. This process is done using best improvement strategy; we choose a pair of edges that results in the lowest possible cost. After updating the current tree and removing selected edges from the sets A and B, algorithm proceeds. This path-relinking algorithm returns the best obtained tree in this process as output.

It is possible that we reach a state that none of the edges that we can remove from current tree have an alternative edge in B. In this situation, we stop the algorithm and report the best solution so far. Algorithm 6 presents our path-relinking approach.

Algorithm 6: Path-relinking-MRCST (T_{start}, T_{end})
Input: trees T_{start}, T_{end}
Output: tree T_*

```
 1. $A \leftarrow T_{start} - T_{end}$
 2. $B \leftarrow T_{end} - T_{start}$
 3. **if** $C(T_{start}) < C(T_{end})$ **then** $T_* \leftarrow T_{start}$ **else** $T_* \leftarrow T_{end}$
 4. $T \leftarrow T_{start}$
 5. $b \leftarrow 1$
 6. **while** $b = 1$ **do**
 7.   $c_{min} \leftarrow \infty$
 8.   $b \leftarrow 0$
 9.   **foreach** edge $e \in A$ **do**
10.     $T = T - \{e\}$
11.     Find set $F \subset B$ of edges such that each of them joins components of $T$
12.     **foreach** edge $f \in F$ **do**
13.       **if** $C(T \cup \{f\}) < c_{min}$ **then** $c_{min} \leftarrow C(T \cup \{f\}), e_{del} \leftarrow e, e_{ins} \leftarrow f, b \leftarrow 1$
14.     **end**
15.     $T = T \cup \{e\}$
16.   **end**
17.   **if** $b = 1$ **then**
18.     $T = T - \{e_{del}\} \cup \{e_{ins}\}, A = A - \{e_{del}\}, B = B - \{e_{ins}\}$
19.     **if** $C(T) < C(T_*)$ **then** $T_* = T$
20.   **end**
21. **end**
22. **return** $T_*$
```

4.3 GRASP with Path-relinking Algorithm

The method of path-relinking we have used that starts from the better solution is called backward relinking. Ribeiro et al. [16] have observed that often starting the path-relinking from the lower cost solution, yields better results. This is because a lower cost solution has better neighbors around itself compared to a higher cost solution, and if we start the trajectory from a good solution the chance of finding another good solution increases. We have also observed this fact in our preliminary experiments, so we use backward relinking.

Another option that we have in designing our GRASP with Path-relinking algorithm is choosing the method of applying path-relinking. One strategy is to apply path-relinking as a post-optimization step to all members of the pool of elite solutions. The other strategy is to use the path-relinking procedure as an intensification method in each iteration of the GRASP. As mentioned in [14] and as our initial experiments showed, the later method often results in better results; hence we apply path-relinking in each iteration of GRASP after local search step.

Now we describe details of the GRASP with Path-relinking algorithm. We use a fixed size pool for our GRASP with Path-relinking algorithm. In the first iteration, we put the result of GRASP algorithm in this pool. In the next iterations, after local search phase of GRASP finds a solution, we select a random solution form the pool and for these two solutions we run the path-relinking algorithm starting from the lower cost solution. The result of path-relinking procedure is tested for insertion into the pool. If the pool is not full yet, this new solution is inserted into it; otherwise if it is better than the worst solution of the pool, then the new solution replaces it in the pool. The best tree found over all iterations is returned as the result of algorithm. Algorithm 7 illustrates steps of our GRASP with Path-relinking approach.

Algorithm 7: GRASP-Path-relinking-MRCST
Output: tree T_*

```
 1. $T_*$ ← A random spanning tree
 2. $P \leftarrow \emptyset$
 3. while stopping criteria is not met do
 4.   $T_1$ ← RandomizedGreedy-MRCST
 5.   $T_2$ ← LocalSearch-MRCST ($T_1$)
 6.   if $P \neq \emptyset$ then
 7.     $R$ ← A random member of $P$
 8.     if $C(T_2) < C(R)$ then
 9.       $T_{start} = T_2$
10.       $T_{end} = R$
11.     else
12.       $T_{start} = R$
13.       $T_{end} = T_2$
14.     end
15.     $T$ ← Path-relinking-MRCST ($T_{start}, T_{end}$)
16.   else
17.     $T \leftarrow T_2$
18.   end
19.   if $|P| < p$ then
20.     $P \leftarrow P \cup \{T\}$
21.   else
22.     Find $B \in P$ such that $\forall D \in P \cdot C(B) \geq C(D)$
23.     if $C(T) < C(B)$ then $P \leftarrow P \cup \{T\} - \{B\}$
24.   end
25.   if $C(T) < C(T_*)$ then $T_* = T$
26. end
27. return $T_*$
```

This is our final algorithm. It remains to specify the pool size and the stopping criteria. We give the value of pool size and the conditions for stopping the main loop of algorithm in the next section.

5. Experimental Results

In order to show effectiveness of the proposed approach, we executed Algorithm 7 on a set of benchmark instances proposed by Julstrom [10]. Furthermore, they were used by Singh [17] and Singh and Sundar [18]. This dataset consists of 35 instances and it includes two groups of graphs: Euclidean graphs, and random graphs.

Euclidean graphs are obtained from OR-Library [22]; they were proposed for the Steiner tree problem. Each graph consists of a set of points in a unit square in the plane. These points are considered as vertices of a complete graph. The group contains seven Euclidean graphs for each size of 50, 100, and 250 vertices.

Random graphs are complete graphs that were generated specially for this problem by Julstrom [10]. This group consists of 14 graphs and it has seven random graphs for each size of 100, and 300 vertices. The weights of the edges between vertices of these graphs are chosen randomly from the interval [0.01, 0.99].

We implemented our algorithm using C++ language under Microsoft Visual Studio, on a 2.8 GHz Pentium 4 Windows XP machine with 512 MB RAM. We executed the algorithm for 30 runs on each instance of the dataset; other authors [10, 17, 18] have also used this number of runs.

The only parameter of our GRASP with path-relinking algorithm (GRASP+PR) is the pool size that we set it to be $4\sqrt{n}$. We found this value by performing some initial experiments. We set the stopping criteria to be a time limit of 40 seconds for graphs having at most 100 vertices, and 300 seconds for larger graphs. In addition, if the algorithm reaches the best known value of the routing cost of the input graph, it stops.

Results of our method on these benchmark problems were compared to the perturbation based local search (PB-LS) of Singh [17], and the artificial bee colony with local search (ABC+LS) of Singh and Sundar [18]. Both of these approaches have better results than the one of [10], and so we did not include results of [10] in this comparison. In tables 1 to 4, we present results of our algorithm and results of ABC+LS, and PB-LS algorithms. In these tables, we report the best value, mean value, standard deviation, and average execution time for the algorithms.

In Table 1, for each instance, the best value found in 30 runs is reported. In each row of this table, the smallest best values are shown in boldface. On Euclidean graphs, our algorithm always finds the best value, except for the e250.6 instance. Moreover, on six other graphs of size 250, our results are better than the ones due to the other algorithms. Results of ABC+LS are similar to ours on graphs of size 100 and 50. Algorithm PB-LS is successful on 50 vertices graphs and some 100 vertices graphs. On random graphs, the three algorithms have similar results for all instances.

Table 1. Best values found by algorithms on Euclidean (e) and random (r) graphs

	Algorithms		
Instance	PB-LS	ABC+LS	GRASP+PR
e50.1	**983.5**	**983.5**	**983.5**
e50.2	**901.3**	**901.3**	**901.3**
e50.3	**888.3**	**888.3**	**888.3**
e50.4	**776.9**	**776.9**	**776.9**
e50.5	**847.9**	**847.9**	**847.9**
e50.6	**818.1**	**818.1**	**818.1**
e50.7	**865.6**	**865.6**	**865.6**
e100.1	**3507.0**	**3507.0**	**3507.0**
e100.2	3308.0	**3307.9**	**3307.9**
e100.3	**3566.3**	**3566.3**	**3566.3**
e100.4	3448.2	**3448.1**	**3448.1**
e100.5	3637.7	**3637.0**	**3637.0**
e100.6	3437.6	**3436.5**	**3436.5**
e100.7	3703.7	**3703.5**	**3703.5**
e250.1	22137.4	22089.6	**22087.9**
e250.2	22797.9	22775.2	**22770.7**
e250.3	21888.8	21886.1	**21871.2**
e250.4	23456.6	23428.5	**23422.7**
e250.5	22420.1	22386.9	**22378.4**
e250.6	22312.6	**22285.3**	22290.3
e250.7	22936.5	22923.9	**22908.8**
r100.1	**597.9**	**597.9**	**597.9**

r100.2	**586.0**	**586.0**	**586.0**
r100.3	**607.0**	**607.0**	**607.0**
r100.4	**598.4**	**598.4**	**598.4**
r100.5	**624.4**	**624.4**	**624.4**
r100.6	**615.5**	**615.5**	**615.5**
r100.7	**514.7**	**514.7**	**514.7**
r300.1	**4131.1**	**4131.1**	**4131.1**
r300.2	**4040.7**	**4040.7**	**4040.7**
r300.3	**4134.8**	**4134.8**	**4134.8**
r300.4	**4229.3**	**4229.3**	**4229.3**
r300.5	**3951.9**	**3951.9**	**3951.9**
r300.6	**4314.4**	**4314.4**	**4314.4**
r300.7	**4093.9**	**4093.9**	**4093.9**

Table 2 shows mean values of the algorithms in 30 runs. On Euclidean instances, algorithm GRASP+PR finds the best mean value on all of the instances, except for the e250.6 instance, where result due to ABC+LS is better. On other instances, our algorithm is clearly superior. Algorithm ABC+LS for two, and algorithm PB-LS for three instances have a similar result to ours. On random graphs ABC+LS reaches the same mean values as the ones for GRASP+PR, except for the r300.6 instance, and PB-LS reaches the best mean value for six 100 vertices graphs.

Table 2. Mean values found by algorithms on Euclidean (e) and random (r) graphs

	Algorithms		
Instance	PB-LS	ABC+LS	GRASP+PR
e50.1	983.6	983.6	**983.5**
e50.2	901.5	**901.3**	**901.3**
e50.3	**888.3**	888.7	**888.3**
e50.4	777.0	**776.9**	**776.9**
e50.5	**847.9**	848.0	**847.9**
e50.6	**818.1**	818.2	**818.1**
e50.7	865.7	866.1	**865.6**
e100.1	3510.4	3507.9	**3507.2**
e100.2	3310.0	3308.7	**3307.9**
e100.3	3567.7	3566.5	**3566.3**
e100.4	3451.3	3451.0	**3448.3**
e100.5	3643.3	3639.4	**3637.3**
e100.6	3442.0	3438.0	**3437.3**
e100.7	3706.4	3704.6	**3703.9**
e250.1	22199.2	22144.8	**22125.6**
e250.2	22970.8	22838.6	**22783.6**
e250.3	22069.4	21927.7	**21895.5**
e250.4	23581.4	23467.6	**23440.4**
e250.5	22492.9	22423.8	**22395.0**
e250.6	22397.2	**22314.2**	22314.7
e250.7	23003.3	22966.2	**22941.7**
r100.1	**597.9**	**597.9**	**597.9**
r100.2	**586.0**	**586.0**	**586.0**
r100.3	**607.0**	**607.0**	**607.0**

r100.4	602.1	**598.4**	**598.4**
r100.5	**624.4**	**624.4**	**624.4**
r100.6	**615.5**	**615.5**	**615.5**
r100.7	**514.7**	**514.7**	**514.7**
r300.1	4196.1	**4131.1**	**4131.1**
r300.2	4131.2	**4040.7**	**4040.7**
r300.3	4220.6	**4134.8**	**4134.8**
r300.4	4272.6	**4229.3**	**4229.3**
r300.5	4041.6	**3951.9**	**3951.9**
r300.6	4458.8	4314.5	**4314.4**
r300.7	4299.4	**4093.9**	**4093.9**

Table 3 compares standard deviation of the three algorithms on each benchmark instance. On Euclidean graphs, our algorithm always has the lowest standard deviation, and only in some 50 vertices graphs, algorithms PB-LS and ABC+LS reach the same values. On random graphs, ABC+LS and GRASP+PR have the best standard deviations, but PB-LS reaches best values only on six instances.

Table 3. Standard deviation of algorithms on Euclidean (e) and random (r) graphs

	Algorithms		
Instance	PB-LS	ABC+LS	GRASP+PR
e50.1	0.2	0.1	**0.0**
e50.2	0.1	**0.0**	**0.0**
e50.3	0.1	0.4	**0.0**
e50.4	0.4	**0.0**	**0.0**
e50.5	**0.0**	**0.0**	**0.0**
e50.6	**0.0**	**0.0**	**0.0**
e50.7	0.2	0.5	**0.0**
e100.1	2.7	1.1	**0.5**
e100.2	2.0	1.1	**0.0**
e100.3	0.9	0.8	**0.0**
e100.4	4.4	2.2	**0.3**
e100.5	6.4	1.9	**0.4**
e100.6	2.3	2.7	**0.9**
e100.7	2.6	2.0	**1.1**
e250.1	77.1	33.0	**19.5**
e250.2	117.4	54.6	**17.2**
e250.3	161.2	16.5	**13.7**
e250.4	101.3	19.9	**12.5**
e250.5	50.7	30.0	**8.2**
e250.6	86.6	22.3	**9.8**
e250.7	34.6	30.4	**16.2**
r100.1	**0.0**	**0.0**	**0.0**
r100.2	**0.0**	**0.0**	**0.0**
r100.3	**0.0**	**0.0**	**0.0**
r100.4	3.0	**0.0**	**0.0**
r100.5	**0.0**	**0.0**	**0.0**
r100.6	**0.0**	**0.0**	**0.0**
r100.7	**0.0**	**0.0**	**0.0**

r300.1	91.3	**0.0**	**0.0**
r300.2	138.2	**0.0**	**0.0**
r300.3	82.5	**0.0**	**0.0**
r300.4	31.6	**0.0**	**0.0**
r300.5	69.8	**0.0**	**0.0**
r300.6	52.9	**0.0**	**0.0**
r300.7	159.7	**0.0**	**0.0**

In Table 4, we reported average execution times in seconds for the algorithms. Running environments of the algorithms have been different. We executed our algorithm on a 2.8 GHz Pentium 4 Windows XP machine, but algorithms PB-LS and ABC+LS were executed on a 3.0 GHz Pentium 4 under Red Hat Linux 9.0. Moreover, PB-LS runs its main loop for 50000 times and ABC+LS after 20n non-improving iterations, executes a local search algorithm. These termination conditions differ from ours. Due to these facts, we are not able to exactly compare the running times. However, it can be seen that in most instances GRASP+PR has less running time, specially for random graphs.

Table 4. Average execution time of the algorithms on Euclidean (e) and random (r) graphs (in seconds)

	Algorithms		
Instance	PB-LS	ABC+LS	GRASP+PR
e50.1	7.5	4.7	0.3
e50.2	7.9	3.6	0.5
e50.3	7.6	4.4	0.7
e50.4	8.2	3.2	0.6
e50.5	8.7	3.2	0.1
e50.6	8.2	4.1	0.1
e50.7	8	4.1	0.2
e100.1	54.7	29.7	16.9
e100.2	53.9	28.9	9.4
e100.3	60.6	25.1	5.0
e100.4	53.5	25.1	26.5
e100.5	50.5	29.5	31.8
e100.6	56.5	28.9	35.6
e100.7	55.4	28.5	13.8
e250.1	590.5	266.9	290.0
e250.2	573.3	277.2	269.6
e250.3	590.7	303	256.9
e250.4	573	309.5	259.9
e250.5	563.3	296.5	272.2
e250.6	605	377	298.5
e250.7	585.4	321	286.6
r100.1	52.9	16.3	0.3
r100.2	54.5	16.3	0.2
r100.3	52.6	11.1	0.2
r100.4	51.6	16.5	0.3
r100.5	54.9	16.5	0.2
r100.6	54.4	16	0.2
r100.7	53.1	14.8	0.2
r300.1	709.1	472.4	13.5
r300.2	674.3	307.9	9.7

r300.3	697.1	467.8	13.0
r300.4	668.2	364.7	11.7
r300.5	681.1	272.6	5.4
r300.6	681.3	600	12.8
r300.7	689.4	383.5	12.1

6. Conclusion

We presented a GRASP with path-relinking algorithm for the MRCST problem. The computational results on a set of benchmark problems showed the effectiveness of our algorithm. This fast algorithm improved the best known values for some of the benchmark instances and compared to the best published algorithms, the average quality of solutions was better. In addition, our algorithm resulted in lower standard deviations than the ones corresponding to other algorithms.

References

[1] Ahuja, R.K. and Murty, V.V.S. (1987), Exact and heuristic algorithms for the optimum communication spanning tree problem, *Transportation Science*, 21, 163-170.

[2] Bresina, J.L. (1996), Heuristic-biased stochastic sampling. *Proceedings of the 13th National Conference on Artificial Intelligence*, AAAI Press: Portland, pp. 271–278.

[3] Campos, R. and Ricardo, M. (2008), A fast algorithm for computing minimum routing cost spanning trees, *Computer Networks*, 52, 3229-3247.

[4] Dijkstra, E.W. (1959), A note on two problems in connexion with graphs, *Numerische mathematik*, 1, 269-271.

[5] Feo, T.A. and Resende, M.G.C. (1989), A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8, 67-71.

[6] Fischetti, M., Lancia, G. and Serafini, P. (2002), Exact algorithms for minimum routing cost trees, *Networks*, 39, 161-173.

[7] Glover, F. (1996), Tabu search and adaptive memory programming-advances, applications and challenges. In: Barr, R.S. Helgason, R.V. and Kennington, J.L. (Eds), Interfaces in Computer Science and Operations Research, Springer, US, 1, pp. 1-75

[8] Hu, T.C. (1974), Optimum communication spanning trees, *SIAM Journal on Computing*, 3, 188-195.

[9] Johnson, D.S., Lenstra, J.K. and Rinnooy Kan, A.H.G. (1978), The complexity of the network design problem, *Networks*, 8, 279-285.

[10] Julstrom, B.A. (2005), The blob code is competitive with edge-sets in genetic algorithms for the minimum routing cost spanning tree problem, *Proceeding of the 2005 conference on Genetic and evolutionary computation*, ACM, pp. 585-590.

[11] Laguna, M. and Marti, R. (1999), GRASP and path relinking for 2-layer straight line crossing minimization, *INFORMS Journal on Computing*, 11, 44-52.

[12] Picciotto, S. (1999), How to Encode a Tree, Ph.D. thesis, University of California, San Diego.

[13] Prim, R.C. (1957), Shortest connection networks and some generalizations, *Bell System Technical Journal*, 36, 1389-1401.

[14] Resende, M.G.C. and Ribeiro, C.C. (2005), GRASP with path-relinking: recent advances and applications, In: Ibaraki, T. Nonobe, K. and Yagiura, M. (Eds), Metaheuristics: progress as real problem solvers, Springer US, pp. 29-63.

[15] Resende, M.G.C. and Ribeiro, C.C. (2010), Greedy randomized adaptive search procedures: advances, hybridizations, and applications, In: Gendreau M. and Potvin J. (Eds), Handbook of Metaheuristics, Springer US, pp. 283-319.

[16] Ribeiro, C.C., Uchoa, E. and Werneck. R.F. (2002), A hybrid GRASP with perturbations for the Steiner problem in graphs, *INFORMS Journal on Computing*, 14, 228-246.

[17] Singh, A. (2008), A new heuristic for the minimum routing cost spanning tree problem, *Proceeding of the International Conference on Information Technology (ICIT'08)*, IEEE, pp. 9-13.

[18] Singh, A. and Sundar, S. (2011), An artificial bee colony algorithm for the minimum routing cost spanning tree problem, *Soft Computing*, 15, 2489-2499.

[19] Wong, R.T. (1980), Worst-case analysis of network design problem heuristics, *SIAM Journal on Algebraic Discrete Methods*, 1, 51-63.

[20] Wu, B.Y., Chao, K.M. and Tang, C.Y. (2000), Approximation algorithms for the shortest total path length spanning tree problem, *Discrete Applied Mathematics*, 105, 273-289.

[21] Wu, B.Y., Lancia, G., Bafna, V., Chao, K.M., Ravi, R. and Tang, C.Y. (1999), A polynomial-time approximation scheme for minimum routing cost spanning trees, *SIAM Journal on Computing*, 29, 761-778.

[22] Beasley J., OR-Library, http://people.brunel.ac.uk/~mastjjb/jeb/info.html, 2005.