Soft Comput (2011) 15:2489–2499
DOI 10.1007/s00500-011-0711-6

ORIGINAL PAPER

An artificial bee colony algorithm for the minimum routing cost spanning tree problem

Alok Singh · Shyam Sundar

Published online: 28 April 2011

Abstract Given a connected, weighted, and undirected graph, the minimum routing cost spanning tree problem seeks a spanning tree of minimum routing cost on this graph, where routing cost of a spanning tree is defined as the sum of the costs of the paths connecting all possible pairs of distinct vertices in that spanning tree. This problem has several important applications in networks design and computational biology. In this paper, we have proposed an artificial bee colony (ABC) algorithm-based approach for this problem. We have compared our approach against four best methods reported in the literature—two genetic algorithms, a stochastic hill climber and a perturbation-based local search. Computational results show the superiority of our ABC approach over other approaches.

Keywords Artificial bee colony algorithm · Constrained optimization · Heuristic · Minimum routing cost spanning tree · Swarm intelligence

1 Introduction

Given a connected, weighted, and undirected graph, the routing cost of any of its spanning tree is defined as the sum of the costs of the paths connecting all possible pairs of distinct vertices in that spanning tree. The minimum routing cost spanning tree (MRCST) problem consists in finding a spanning tree with minimum routing cost among all spanning trees of the graph. It is a NP-Hard problem (Johnson et al. 1978). To find the routing cost of a spanning tree, it is not necessary to enumerate every possible path in that spanning tree. Instead, routing cost can be computed more easily by first determining for each edge of the spanning tree the count of the paths containing that edge, and then summing the product of this count and edge weight for every edge (Julstrom 2005). Formally, let $G = (V, E)$ be a connected undirected graph, where V denotes the set of vertices and E denotes the set of edges. Given a non-negative weight function w: $E \rightarrow \mathbb{R}^+$ associated with its edges, the MRCST problem seeks on this graph a spanning tree $\mathrm{T} \subseteq \mathrm{E}$ that minimizes $W = \sum_{e \in T} c_e.w(e)$, where c_e is the count of those paths in T which contains edge e. c_e can be determined by calculating the product of the number of vertices in the two partitions arising as a result of temporarily removing e from the spanning tree. This is due to the fact that any path connecting a vertex in one partition to a vertex in another has to contain e. Therefore, the total number of paths containing e is simply the product of the number of vertices in one partition with the number of vertices in another partition. Here, it is to be noted that in a spanning tree, exactly one path exists between any pair of distinct vertices.

The MRCST problem has several important applications in communication network design, where the cost of an edge may represent the cost incurred in routing messages between its endpoints. For example, consider a situation (Wu and Chao 2004), where the cost of an edge represents the delay in routing a message between its endpoints and one has to find a spanning tree that minimizes the average delay of communicating between any two vertices via the spanning tree. The delay between any two vertices is the sum of the delays of the edges lying on the unique tree path

A. Singh (✉) · S. Sundar
Department of Computer and Information Sciences,
University of Hyderabad, Hyderabad 500046, India
e-mail: alokcs@uohyd.ernet.in

S. Sundar
e-mail: mc08pc17@uohyd.ernet.in

connecting the two vertices. Clearly, this is an instance of MRCST problem as minimizing the average delay being equivalent to minimizing the total delay between all pair of vertices (Wu and Chao 2004). The problem gains importance in heterogeneous computer networks, where different subnetworks are connected through bridges (Campos and Ricardo 2008). Bridging mandates that the active network topology should be a spanning tree. In such a situation, a MRCST is always an optimal spanning tree from the routing cost point of view provided the probability of communication between any pair of nodes is the same. The MRCST problem also finds application in multiple sequence alignment problems in computational biology (Wu and Chao 2004).

Most of the work on MRCST problem is focused on designing approximation algorithms for it. Wong (1980) proposed an approximation scheme that computes n shortest path trees, where n is the number of vertices in the graph, and returns the tree with smallest routing cost among these trees. Wu et al. (1999) proposed an approximation algorithm that is applicable to metric graphs only. Grout (2005) described an algorithm that gives better results in shorter time on homogeneous graphs in comparison to the approach of Wong, but performs poorly on non-homogeneous graphs. Campos and Ricardo (2008) proposed another algorithm that gives results comparable to Wong's algorithm in lesser time for practical cases. Fischetti et al. (2002) presented an exact branch-and-price algorithm for the MRCST problem.

Though the approximation algorithms provide solutions that are guaranteed to be within a certain factor of optimum, solutions obtained through these algorithms on many problems cannot compete with the solutions provided by the state-of-the-art metaheuristics. Therefore, on many problems, the use of these approximation algorithms is limited mainly to either dynamic environments where a quick solution is desired or in providing an initial solution to a metaheuristic technique.

Among the metaheuristic techniques, Julstrom (2005) proposed two genetic algorithms and a stochastic hill climber for the problem. One genetic algorithm uses Blob code (Picciotto 1999) to represent a spanning tree, whereas other is edge-set-coded, i.e., it represents a spanning tree by the set of its edges (Raidl and Julstrom 2003). The edge-set-coded genetic algorithm uses a crossover operator that is derived from Kruskal's algorithm for finding the minimum spanning tree. The mutation operator for this genetic algorithm replaces, with small probability, each edge with some randomly chosen edge connecting the two partitions resulting from deletion of the original edge. The Blob-coded genetic algorithm uses two point crossover and position-by-position mutation operator. However, Blob code requires a decoder to convert the code into an equivalent spanning tree. Starting from a random initial solution, the stochastic hill climber repeatedly generates a new solution that differs from the current solution on exactly one edge. If this new solution is better than the current solution, then it becomes the current solution, otherwise, it is discarded, and, the whole process is repeated again and again. On the test instances considered, the stochastic hill climber performed best in terms of solution quality followed by edge-set-coded genetic algorithm, though Blob-coded genetic algorithm is fastest among the three followed by stochastic hill climber. Earlier versions of the edge-set-coded genetic algorithm and the stochastic hill climber are appeared in Julstrom (2002) along with an exhaustive hill climber.

Singh (2008) proposed a perturbation-based local search PB-LS that also encodes a spanning tree by the set of its edges. Starting from an initial solution to the MRCST problem, PB-LS repeatedly generates a new solution by applying a local search. This local search randomly deletes an edge and tries all possible edges connecting the two partitions. The edge that yields the routing cost spanning tree of least cost is included into the solution. If the resulting solution is better than the current solution, it becomes the new current solution. After this, the local search is again applied to the current solution. If this procedure fails to improve the best solution for a specific number of iterations then the current solution is perturbed by randomly removing and inserting some edges. The whole process is repeated for a fixed number of iterations. The local search used in PB-LS randomly deletes an edge from the spanning tree and then tests all edges in G connecting the two resulting partitions for possible inclusion. The edge that results in a spanning tree of least routing cost is included in the tree. The newly included edge is prohibited from deletion in the next iteration of the local search because this edge represents the best possible edge connecting the two partitions at that iteration.

In this paper, we have proposed an artificial bee colony (ABC) algorithm-based approach for the MRCST problem. The best solution obtained by the ABC algorithm is improved further through a local search. We have compared our approach against PB-LS (Singh 2008) and the three methods proposed by Julstrom (2005), which are the best methods known so far for the problem. Our approach obtained better quality solutions in comparison to these approaches.

The remainder of this paper is organized as follows: Sect. 2 gives an overview of the ABC algorithm, while Sect. 3 describes our approach for the MRCST problem. Section 4 presents the performance of our ABC approach on a set of 35 benchmark instances and compares it with other approaches. Finally, Sect. 5 provides some concluding remarks.

2 Overview of the ABC algorithm

The ABC algorithm is a population-based new metaheuristic approach motivated by the intelligent foraging behavior of honeybee swarm. The ABC algorithm is proposed by Karaboga (2005) and further developed by Karaboga and Basturk (Basturk and Karaboga 2006; Karaboga and Basturk 2007a, b, 2008, 2009) and Singh (2009). Usually, the foraging bees are divided into three classes—employed, onlookers and scouts. "Employed" bees are those bees that are presently tapping a food source. The employed bees are responsible for bringing loads of nectar from the food sources to the hive and sharing the information about food sources with onlooker bees. "Onlookers" are those bees that are waiting in the hive for the information to be shared by the employed bees about their food sources and "scouts" are those bees that are presently looking for new food sources in the vicinity of the hive. Employed bees share information about food sources by dancing in a common area in the hive called dance area. The nature and duration of a dance depends on the nectar content of the food source currently being tapped by the dancing bee. Onlooker bees observe numerous dances before choosing a food source. The onlookers have a tendency to choose a food source with a probability proportional to the nectar content of that food source. Therefore, good food sources attract more bees than the bad ones. Whenever a bee, whether it is scout or onlooker, finds a food source it becomes employed. Whenever a food source is completely exhausted, all the employed bees associated with it leave it, and can again become scouts or onlookers. So in a way scout bees can be perceived as performing the job of exploration, whereas employed and onlooker bees can be perceived as performing the job of exploitation.

Inspired by this foraging behavior of honeybees, Karaboga (2005) proposed the artificial bee colony (ABC) algorithm. In the ABC algorithm, each food source represents a possible solution to the problem being considered and the nectar amount of a food source corresponds to the fitness of the solution being represented by that food source. In this algorithm also, colony of artificial bees (bees for short) contains the same three types of bees, viz. employed, onlookers and scouts. First half of the bee colony contains employed bees, whereas the latter half consists of onlookers. However, unlike real bee colonies, there is a one-to-one correspondence between the employed bees and the food sources (solution). Every employed bee is associated with a unique food source (solution), i.e., the number of food sources is same as the number of employed bees. The employed bee of an exhausted food source abandons it and becomes a scout. As soon as it finds a new food source it again becomes employed. In the ABC algorithm, the action of a scout bee is modeled by generating a new food source (solution) randomly and associating this scout bee with this food source to make it again employed. We will use food source and solution interchangeably throughout the description of the ABC algorithm.

The ABC algorithm follows an iterative process. It begins by associating all employed bees with randomly generated food sources (solution). Then at each iteration, each employed bee determines a food source in the neighborhood of its currently associated food source and evaluates its nectar amount (fitness). If its nectar amount is better than that of the food source to which it is currently associated then that employed bee shifts to this new food source abandoning the old one, otherwise it remains at the old food source. This phase where every employed bee determines a neighboring food source can be termed as "employed bee phase". Once this phase is over, "onlooker bee phase" begins where all employed bees share the nectar information of their corresponding food sources with the onlookers. Onlookers select food sources probabilistically according to their fitness. The probability p_i of selecting a food source i is determined using the following expression:

$$p_i = \frac{F_i}{\sum_{j=1}^{m} F_j}$$

where F_i is the fitness of the solution corresponding to the food source i and m is the total number of food sources. In the genetic algorithm parlance this selection scheme is called "roulette wheel" selection. Clearly, with this scheme, higher the fitness of a food source, higher will be the probability of its selection. As a result good food sources will get more onlookers than the bad ones. After all onlookers have selected their food sources, each of them determines a food source in the neighborhood of their selected food source in the manner similar to employed bee phase and computes its fitness. Among all the neighboring food sources determined by the onlookers associated with a particular food source i and the food source i itself, the best food source is determined. This best food source becomes the new location of the food source i in the next iteration. Once the new locations of all food sources are determined, the onlooker bee phase ends and the next iteration of the ABC algorithm begins. The whole process is repeated until the termination condition is satisfied. Hence, each iteration of ABC algorithm consists of two phases, viz. employed bee phase and onlooker bee phase. If a solution corresponding to a particular food source does not improve for a predetermined number of iterations then that food source is assumed to be exhausted and its associated employed bee abandons it to become scout. A new food source (solution) is randomly generated to handle this scout. This scout bee is associated with the newly generated food source and its status is again changed from scout to employed.

In essence, the ABC algorithm is based on the fact that in the neighborhood of a good solution, chances of finding even better solutions are high. That is why more onlookers are deputed for good solutions so that their neighborhood can be explored more thoroughly in comparison to poorer solutions. Though in the employed bee phase every solution is given a chance to improve itself, the onlooker bee phase is biased towards good solutions which get more chances to improve themselves. However, if a solution is locally optimal with respect to the whole neighborhood then it cannot be improved. Therefore, any attempt to improve it is futile. This is where the concept of scout comes to our rescue. As it is computationally expensive to determine whether a solution is locally optimal or not, the ABC algorithm assumes that if a solution does not improve for certain number of iterations then it is locally optimal, and therefore, replaces this solution with a randomly generated solution. The pseudo-code of ABC algorithm is given in Fig. 1 where n_e and n_o are, respectively, the number of employed and onlooker bees.

Generate_Neighboring_Solution(X) is a function that returns either a solution in the neighborhood of the solution X or $\varnothing$ if it fails to find a neighboring solution. Exact implementation of this function depends on the problem in consideration. Select_and_Return_Index($X_1, X_2, \ldots, X_{n_e}$) is another function that selects a solution from solutions $X_1, X_2, \ldots, X_{n_e}$ for an onlooker and returns the index of the solution selected. Exact implementation of this function depends on the selection policy used.

```
procedure ABC
    generate $n_e$ random solutions $E_1, E_2, \ldots, E_{n_e}$
    best_sol := best solution among $E_1, E_2, \ldots, E_{n_e}$
    repeat
        for i:=1 to $n_e$ do
            $E'$ := Generate_Neighboring_Solution($E_i$)
            if ($E' == \varnothing$) then
                replace $E_i$ with a random solution
            else if ($E'$ is better than $E_i$) then
                $E_i := E'$
            else if ($E_i$ has not changed over last $it_{noimp}$ iterations) then
                replace $E_i$ with a random solution
            if ($E_i$ is better than best_sol) then
                best_sol := $E_i$
        end for
        for i:=1 to $n_o$ do
            $p_i$ := Select_and_Return_Index($E_1, E_2, \ldots, E_{n_e}$)
            $S_i$ := Generate_Neighboring_Solution($E_{p_i}$)
            if ($S_i == \varnothing$) then
                artificially assign fitness worse than $E_{p_i}$ to $S_i$
            if ($S_i$ is better than best_sol) then
                best_sol := $S_i$
        end for
        for i:=1 to $n_o$ do
            if ($S_i$ is better than $E_{p_i}$) then
                $E_{p_i} := S_i$
        end for
    until (termination condition is satisfied)
    return best_sol
end procedure
```

Fig. 1 Pseudo-code of the ABC algorithm

The process of determination of a food source in the neighborhood of a particular food source depends on the nature of the problem. Karaboga's original ABC algorithm was proposed for continuous optimization. In his model, the food source in the neighborhood of a particular food source is determined by altering the value of one randomly chosen solution parameter and keeping other parameters unchanged. This is done by adding to the current value of the chosen parameter the product of a uniform variate in $[-1, 1]$ and the difference in values of this parameter for this food source and some other randomly chosen food source. This method cannot be applied for discrete optimization problems for which it produces at best a random effect. Later, Singh (2009) presented a method that is applicable for the subset selection problems. To generate a neighboring solution, in this method, an object is randomly removed from the solution and in its place another object, which is not already present in the solution is inserted. The object to be inserted is selected from another randomly chosen solution. If there are more than one candidate objects for insertion then ties are broken arbitrarily. This method is based on the idea that if an object is present in one good solution then it is highly likely that this object is present in many good solutions. Another advantage of this method is that it helps in keeping a check on the number of duplicate solutions. If the method fails to find an object in the randomly chosen solution different from the objects in the original solution then that means that the two solutions are identical. Such a situation was called "collision" and it is resolved by making the employed bee associate with the original solution scout. This eliminates one duplicate solution.

3 The ABC approach to MRCST problem

Our approach (ABC + LS) is a combination of an ABC algorithm and a local search heuristic. The best solution obtained through the ABC algorithm is improved further by using the local search.

3.1 The ABC algorithm

The main features of our ABC algorithm for the MRCST problem are as follows.

3.1.1 Solution encoding

We have used edge-set encoding (Raidl and Julstrom 2003) to represent a spanning tree. Edge-set encoding represents a spanning tree by the set of its $n-1$ edges, where n is the number of vertices in the graph. Approaches proposed in Julstrom (2005) and Singh (2008) also use this encoding.

3.1.2 *Initial employed bee solutions*

The algorithm is initialized by assigning a randomly generated solution to every employed bee. Starting from a random start vertex, each initial solution is generated in a manner similar to Prim's algorithm (Prim 1957) for generating a spanning tree. However, at each stage, instead of selecting a least cost edge among all edges connecting a vertex in the partially constructed tree to a vertex not in the partially constructed tree, it selects an edge using roulette wheel selection from all candidate edges where the probability of the selection of an edge is inversely proportional to either its weight or square of its weight. With probability p_{sq}, we generate the entire spanning tree by setting the probability of the selection of an edge inversely proportional to square of its weight and with probability $(1-p_{\mathrm{sq}})$, we generate the entire spanning tree by setting the probability of the selection of an edge inversely proportional to its weight.

3.1.3 *Probability of selecting a food source*

Instead of using the usual roulette wheel selection scheme as described in Sect. 2, we have used binary tournament selection for selecting a food source for an onlooker. In the binary tournament selection, two food sources are randomly chosen and better of the two food sources are selected with probability p_{better} and worse of the two with probability $(1-p_{\mathrm{better}})$. We have tried the roulette wheel selection also, but binary tournament selection always gave better results.

3.1.4 *Determination of a food source in the neighborhood of a food source*

The method used here is derived from the method used in Singh (2009). It is based on the concept that if an edge is present in one good solution then it is highly likely that the same edge is present in many good solutions. In order to generate a neighboring solution, this method randomly deletes an edge from the solution, thereby, creating two partitions. Another employed bee solution is chosen randomly and all edges from this solution connecting the two partitions which are different from the deleted edge are tried one-by-one for insertion. The edge that results in routing cost spanning tree (RCST) of least cost is selected for insertion. If we are not able to find any edge in the randomly chosen solution different from the deleted edge connecting the two partitions then the deleted edge is reinserted into the solution and the method starts afresh. Note that, there exist $O(n^2)$ edges connecting the two partitions and our method tries at the maximum only $n-1$ edges.

Normally, the number of the edges that are tried for insertion is much less than $n-1$. Therefore, our method, in effect, curtails the search space from $O(n^2)$ to $O(n)$.

If the method fails for consecutive t_k trials, while generating a neighboring solution for an employed bee, then instead of again starting afresh, we simply make the corresponding employed bee a scout. Actually, repeated failures of the method to find an edge, different from the one deleted, indicates the lack of diversity in the employed bee solutions and therefore, the employed bee is made a scout to increase the diversity. However, we keep on trying in case we are generating a neighboring solution for an onlooker, because there is no point in generating a solution for the onlooker randomly as for the survival this randomly generated solution has to compete with the original solution as well as with the solutions of all those onlookers which are associated with the same original solution. Hence, it is highly likely that such a randomly generated solution dies immediately. This is analogous to the concept of "collision" introduced in Singh (2009). The pseudocode for determining a neighboring food source is given in Fig. 2, where Find_Candidate_Edges(X, Y, Z) is a function that returns all the edges of the solution Z, which can connect the two partitions of the solution X resulting from deletion of the edge Y and are different from edge Y.

3.1.5 *Cost evaluation*

As described in Sect. 1, the routing cost of a spanning tree can be computed by first determining for each edge

```
function Generate_Neighboring_Solution(E)
    E' := E
    i := 0
    nosolution := true
    repeat
        randomly delete an edge e of E'
        randomly select another solution F different from E
        C= Find_Candidate_Edges(E', e, F)
        if (C == ∅) then begin
                i := i + 1
                reinsert the deleted edge e to E'
        end
        else begin
                nosolution := false
                find the edge e' in C that will result in RCST of least cost
                add e' to E'
        end
    until ((nosolution == false) or (i == t_k))
    if (nosolution == true) then
        E' := ∅
    return E'
end function
```

Fig. 2 Method for determining a neighboring food source

e of the spanning tree the count c_e of the paths containing e, and then summing the product of this count and edge weight for every edge. The computation of these c_e values has to be efficient because every time a new solution is created we have to recompute c_e values from scratch. Even a change of one edge can drastically change c_e values. This is true in case of neighboring solutions, which differ on only one edge from an already existing solution. It was suggested in Julstrom (2005) that these c_e values, and hence, routing cost can be determined by doing a traversal of the spanning tree beginning at any node and keeping track of the number of nodes in each subtree. If the number of nodes in a subtree is known, then the number of nodes in the rest of the spanning tree is also known, and, the product of these two values gives the value of c_e for the edge joining that particular subtree to the rest of the spanning tree. Therefore, the routing cost of a spanning tree can be computed in $O(n)$ time provided adjacency list representation is used (Aho et al. 1983) as there are only $n - 1$ edges in a spanning tree.

3.1.6 Other features

Unlike the usual practice, we have used different number of employed bees and onlooker bees. If the solution associated with an employed bee does not improve for it_{noimp} number of iterations then it becomes a scout. Like Singh (2009), here also there is a second possibility in which an employed bee can become scout as described previously while discussing the method for generating a neighboring solution. Also, there is no upper limit on the number of scouts in a single iteration. The number of scouts in a particular iteration depends on the number of times the aforementioned two conditions hold. The solution for a scout is generated in the same manner as the initial solutions.

3.2 The local search

The best routing cost spanning tree obtained through ABC algorithm is further improved by applying a local search. It is an iterative approach. At each iteration the local search deletes each edge of the spanning tree one-by-one and examines all graph edges connecting the two resulting partitions for a possible inclusion. The edge that results in a spanning tree of least routing cost will be selected for inclusion. The local search is applied repeatedly until a complete iteration fails to improve the solution. We have also tried this local search inside the ABC algorithm, but it had made the resulting approach too slow to be of any practical use.

4 Computational results

The ABC + LS has been implemented in C and has been executed on a Pentium 4 system with 512 MB RAM running at 3.0 GHz under Red Hat Linux 9.0. We have used a colony of 200 bees. 50 of these bees are employed, whereas the remaining bees are onlookers, i.e., $n_e = 50$ and $n_o = 150$. In all our experiments with ABC + LS, we have used $p_{\text{better}} = 0.95$, $p_{\text{sq}} = 0.25$, $it_{\text{noimp}} = 5n$ and $t_k = 5$. On a problem instance of size n, ABC component of ABC + LS terminates when the best solution does not improve over $20n$ iterations. All these parameter values are chosen empirically after large number of trials though they are in no way optimal for all instances. To test the effectiveness of ABC alone, we have also implemented a version of our approach where local search is not used. We call this approach PABC (pure ABC). Except for the use of local search, PABC is same as ABC + LS.

To test ABC + LS and PABC, we have used the same 35 test instances as used in Julstrom (2005) and Singh (2008). Out of these 35 instances, 21 are Euclidean, whereas the rest are randomly generated. The Euclidean instances were originally designed for the Euclidean Steiner tree problem. These instances consist of points in the unit square. These points can be considered as the vertices of a complete graph, whose edge-weights are the Euclidean distances between them. These instances are available from Beasley's OR-library.[1] There are 15 Euclidean instances for each of 50, 100 and 250 vertices and first 7 instances of each size were used in Julstrom (2005) and Singh (2008). The random instances were generated by the Julstrom (2005). There are 7 random instances for each of 100 and 300 vertices. The edge-weights of these random instances are uniformly distributed in [0.01, 0.99]. Each Euclidean instance has the name of the form e$n.i$, where n is the number of vertices in the instance and i is its number. Similarly, each random instance has the name of the form r$n.i$. ABC + LS, PABC was executed 30 times on each instance.

We have compared ABC + LS and PABC with edge-set-coded genetic algorithm (Julstrom 2005), Blob-coded genetic algorithm (Julstrom 2005), stochastic hill climber (Julstrom 2005) and PB-LS (Singh 2008). Hereafter, edge-set-coded genetic algorithm, Blob-coded genetic algorithm, stochastic hill climber will be, respectively, referred as ESCGA, BCGA and SHC. Table 1 reports the results of ABC + LS and PABC along with those of ESCGA, BCGA, SHC and PB-LS on 21 Euclidean instances, whereas Table 2 does the same for 14 random instances. Data for ESCGA, BCGA and SHC are taken from Julstrom (2005), whereas data for PB-LS are taken from Singh

[1] http://people.brunel.ac.uk/~mastjjb/jeb/info.html

Table 1 Results of ESCGA, BCGA, SHC, PB-LS, PABC and ABC + LS on 21 Euclidean instances

Inst.	ESCGA			BCGA			SHC			PB-LS			PABC			ABC + LS		
	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD
e50.1	984.8	998.1	17.1	987.6	1,008.1	15.6	985.1	1,003.5	23.9	**983.5**	**983.6**	0.2	**983.5**	**983.6**	0.1	**983.5**	**983.6**	0.1
e50.2	901.4	907.8	6.7	902.3	912.2	12.5	902.0	912.6	13.3	**901.3**	901.5	0.1	**901.3**	**901.3**	0.0	**901.3**	**901.3**	0.0
e50.3	**888.3**	913.1	21.8	889.9	935.5	31.7	**888.3**	908.7	18.4	**888.3**	**888.3**	0.1	**888.3**	888.9	0.4	**888.3**	888.7	0.4
e50.4	778.2	793.8	18.8	777.1	797.3	17.4	**776.9**	792.5	18.5	**776.9**	777.0	0.4	**776.9**	**776.9**	0.3	**776.9**	**776.9**	0.0
e50.5	847.9	858.3	15.1	848.1	865.5	20.4	**847.9**	860.8	30.2	**847.9**	**847.9**	0.0	**847.9**	848.0	0.0	**847.9**	848.0	0.0
e50.6	818.4	825.5	6.1	819.3	843.3	34.4	**818.1**	829.0	22.7	**818.1**	**818.1**	0.0	**818.1**	818.2	0.0	**818.1**	818.2	0.0
e50.7	**865.6**	881.5	14.8	865.9	889.1	16.6	865.9	886.3	16.7	**865.6**	**865.7**	0.2	**865.6**	866.2	0.5	**865.6**	866.1	0.5
e100.1	3,538.4	3,585.5	56.9	3,525.5	3,592.8	65.0	3,513.2	3,553.5	47.1	**3,507.0**	3,510.4	2.7	**3,507.0**	3,508.6	1.4	**3,507.0**	**3,507.9**	1.1
e100.2	3,315.2	3,400.2	48.5	3,342.7	3,407.3	46.3	3,310.6	3,359.7	52.8	3,308.0	3,310.0	2.0	**3,307.9**	3,309.0	1.2	**3,307.9**	**3,308.7**	1.1
e100.3	3,576.0	3,641.8	56.7	3,573.9	3,665.6	76.1	3,566.9	3,610.7	38.4	**3,566.3**	3,567.7	0.9	**3,566.3**	3,566.5	0.8	**3,566.3**	**3,566.5**	0.8
e100.4	3,464.5	3,541.3	57.0	3,468.1	3,551.2	57.6	3,458.4	3,504.0	39.9	3,448.2	3,451.3	4.4	**3,448.1**	3,451.4	2.5	**3,448.1**	**3,451.0**	2.2
e100.5	3,652.8	3,764.3	83.7	3,641.9	3,737.8	63.0	3,639.9	3,707.6	62.9	3,637.7	3,643.3	6.4	**3,637.0**	3,640.4	2.3	**3,637.0**	**3,639.4**	1.9
e100.6	3,455.0	3,487.1	19.9	3,443.3	3,501.3	37.5	3,436.8	3,461.3	18.8	3,437.6	3,442.0	2.3	**3,436.5**	3,438.4	2.7	**3,436.5**	**3,438.0**	2.7
e100.7	3,730.1	3,783.5	61.2	3,733.8	3,819.5	76.2	3,711.6	3,741.8	29.0	3,703.7	3,706.4	2.6	**3,703.5**	3,704.9	2.4	**3,703.5**	**3,704.6**	2.0
e250.1	22,545.7	23,225.3	556.1	22,543.0	23,013.6	306.9	22,177.2	22,568.3	359.6	22,137.4	22,199.2	77.1	22,110.1	22,163.0	28.9	**22,089.6**	**22,144.8**	33.0
e250.2	23,286.6	24,310.7	590.7	23,149.1	23,834.0	563.1	22,961.9	23,433.1	464.9	22,797.9	22,970.8	117.4	**22,775.2**	22,864.7	56.8	**22,775.2**	**22,838.6**	54.6
e250.3	22,394.7	23,094.2	519.7	22,237.0	22,821.4	420.6	22,055.7	22,473.7	310.7	21,888.8	22,069.4	161.2	21,888.1	21,945.0	24.9	**21,886.1**	**21,927.7**	16.5
e250.4	23,725.8	24,824.2	743.9	23,837.3	24,647.0	607.1	23,598.3	23,903.2	293.8	23,456.6	23,581.4	101.3	23,454.4	23,486.7	17.0	**23,428.5**	**23,467.6**	19.9
e250.5	22,604.0	23,352.8	455.0	22,739.4	23,264.8	306.3	22,458.1	22,880.5	256.3	22,420.1	22,492.9	50.7	22,396.7	22,446.8	33.4	**22,386.9**	**22,423.8**	30.0
e250.6	22,662.0	23,326.6	448.8	22,507.4	23,102.6	407.4	22,334.3	22,559.3	171.1	22,312.6	22,397.2	86.6	22,285.5	22,323.3	25.0	**22,285.3**	**22,314.2**	22.3
e250.7	23,390.0	23,853.7	538.7	23,228.2	23,781.2	328.5	23,039.9	23,226.9	190.5	22,936.5	23,003.3	34.6	22,931.4	22,986.6	31.5	**22,923.9**	**22,966.2**	30.4

Table 2 Results of ESCGA, BCGA, SHC, PB-LS, PABC and ABC + LS on 14 random instances

Inst.	ESCGA			BCGA			SHC			PB-LS			PABC			ABC + LS		
	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD
r100.1	598.5	628.0	33.8	625.8	724.4	86.9	**597.9**	609.8	18.4	**597.9**	**597.9**	0.0	**597.9**	**597.9**	0.0	**597.9**	**597.9**	0.0
r100.2	586.0	640.7	37.1	623.0	730.5	53.2	**586.0**	601.6	26.6	**586.0**	**586.0**	0.0	**586.0**	**586.0**	0.0	**586.0**	**586.0**	0.0
r100.3	607.7	668.9	49.5	678.2	756.5	48.1	**607.0**	637.3	55.8	**607.0**	**607.0**	0.0	**607.0**	**607.0**	0.0	**607.0**	**607.0**	0.0
r100.4	607.3	636.8	20.2	628.3	715.9	66.3	**598.4**	610.9	16.9	**598.4**	602.1	3.0	**598.4**	**598.4**	0.0	**598.4**	**598.4**	0.0
r100.5	628.4	668.4	34.9	641.7	757.1	74.8	**624.4**	643.1	19.1	**624.4**	**624.4**	0.0	**624.4**	**624.4**	0.0	**624.4**	**624.4**	0.0
r100.6	615.6	655.1	30.2	639.1	732.8	54.9	**615.5**	**615.5**	0.0	**615.5**	**615.5**	0.0	**615.5**	**615.5**	0.0	**615.5**	**615.5**	0.0
r100.7	514.9	539.0	24.7	531.3	646.8	66.4	**514.7**	514.8	0.1	**514.7**	**514.7**	0.0	**514.7**	**514.7**	0.0	**514.7**	**514.7**	0.0
r300.1	4,926.3	5,444.7	316.9	5,361.7	6,048.2	618.6	**4,131.1**	4,259.8	119.4	**4,131.1**	4,196.1	91.3	**4,131.1**	4,131.4	0.8	**4,131.1**	**4,131.1**	0.0
r300.2	4,957.5	5,488.0	314.2	5,236.3	6,017.8	413.5	**4,040.7**	4,237.6	180.7	**4,040.7**	4,131.2	138.2	**4,040.7**	4,040.9	0.5	**4,040.7**	**4,040.7**	0.0
r300.3	5,013.1	5,452.4	300.8	5,093.5	5,917.3	396.3	**4,134.8**	4,259.5	87.3	**4,134.8**	4,220.6	82.5	**4,134.8**	4,136.5	2.1	**4,134.8**	**4,134.8**	0.0
r300.4	5,012.4	5,773.5	378.4	5,320.0	6,135.4	472.9	**4,229.3**	4,397.1	163.8	**4,229.3**	4,272.6	31.6	**4,229.3**	4,229.5	0.5	**4,229.3**	**4,229.3**	0.0
r300.5	4,622.4	5,433.1	362.7	5,118.2	6,010.7	423.7	**3,951.9**	4,132.3	177.4	**3,951.9**	4,041.6	69.8	**3,951.9**	**3,951.9**	0.0	**3,951.9**	**3,951.9**	0.0
r300.6	5,259.9	5,641.2	222.3	5,408.8	6,083.7	360.4	**4,314.4**	4,517.6	80.2	**4,314.4**	4,458.8	52.9	**4,314.4**	4,319.3	7.5	**4,314.4**	**4,314.5**	0.0
r300.7	4,868.3	5,468.5	342.1	5,342.6	5,990.2	442.9	**4,093.9**	4,268.5	144.9	**4,093.9**	4,299.4	159.7	**4,093.9**	4,094.1	0.7	**4,093.9**	**4,093.9**	0.0

(2008). These tables report for each instance the best and average solution obtained by each of five methods as well as standard deviation of their solution values. For each instance, overall best and overall best average solution values are shown in boldface in these tables. On Euclidean instances, except for 4 instances of size 50, where average solution quality of PB-LS is better, ABC + LS always obtains solutions of same or better average quality than other methods. The best solution found by ABC + LS is always as good as or better than the other methods. On larger instances of size 250, it performs much better than other methods. As far as the performance on 14 random instances is concerned, ABC + LS, PABC, SHC and PB-LS all obtained the same best solution on every instance but the average solution quality of ABC + LS is as good as or better than the other methods. On instances of size 300 though PB-LS and SHC also obtained the same best solution, they performed much worse than ABC + LS in terms of average solution quality. Standard deviation of solution values for ABC + LS is also less for most of the instances.

As far as the performance of PABC is concerned, barring ABC + LS, it outperforms the remaining 4 methods in terms of both best as well as average solution quality except for 5 Euclidean instances where average solution quality of PB-LS is better than PABC. This shows that PABC is an effective heuristic in itself. ABC + LS, anyway, will perform as good as or better than PABC, because the best solution obtained by the ABC algorithm will serve as the input to the local search.

To test the statistical significance of the results of ABC + LS vis-à-vis other approaches, we have performed *t* tests on ABC + LS in conjunction with each of the other approaches except PABC. Actually, ABC + LS is nothing but PABC with a local search that improves the best solution obtained through PABC. Therefore, improvements, if any, in results of ABC + LS over PABC can never be due to random fluctuations and hence *t* test does not have any significance here. Table 3 shows the *t* test results of ESCGA, BCGA, SHC, PB-LS in conjunction with ABC + LS. For each instance and for each method in conjunction with ABC + LS, this table reports the *t* value and (two-tailed) *p* value. Data for ESCGA, BCGA and SHC are taken from Julstrom (2005), whereas data for PB-LS are taken from Singh (2008). Even if we use the 1% significance criterion (p value ≤ 0.01), except for two instances in case of SHC and six instances in case of PB-LS, results of ABC + LS are statistically significant. In most cases, *p* values are quite small showing the statistical significance of the results obtained through ABC + LS.

Table 4 shows the execution time in seconds for all the six methods. Instead of reporting the execution time on

Table 3 *t* test results of ESCGA, BCGA, SHC, PB-LS in conjunction with ABC + LS

Instance	ESCGA		BCGA		SHC		PB-LS	
	t value	*p* value	*t* value	*p* value	*t* value	*p* value	*t* value	*p* value
e50.1	4.644352	0.000020	8.601876	0.000000	4.560495	0.000027	0.000000	1.000000
e50.2	5.313726	0.000002	4.776141	0.000013	4.653583	0.000019	11.254629	0.000000
e50.3	6.129441	0.000000	8.085607	0.000000	5.952100	0.000000	−3.104930	0.003588
e50.4	4.923676	0.000007	6.421575	0.000000	4.618633	0.000022	1.406829	0.167613
e50.5	3.736121	0.000429	4.698600	0.000017	2.321473	0.023800	–*	–
e50.6	6.554713	0.000000	3.996464	0.000184	2.605905	0.011625	–*	–
e50.7	5.696026	0.000000	7.585487	0.000000	6.622180	0.000000	−2.447880	0.019104
e100.1	7.468424	0.000000	7.153075	0.000000	5.301346	0.000002	4.205791	0.000153
e100.2	10.330666	0.000000	11.660952	0.000000	5.289354	0.000002	2.602924	0.013111
e100.3	7.273263	0.000000	7.132235	0.000000	6.303147	0.000000	3.984496	0.000296
e100.4	8.670623	0.000000	9.521148	0.000000	7.264478	0.000000	0.285538	0.776812
e100.5	8.171198	0.000000	8.551017	0.000000	5.936033	0.000000	3.026252	0.004428
e100.6	13.391462	0.000000	9.221685	0.000000	6.719321	0.000000	4.195810	0.000157
e100.7	7.057558	0.000000	8.256123	0.000000	7.009309	0.000000	2.285097	0.027984
e250.1	10.623538	0.000000	15.416553	0.000000	6.423523	0.000000	3.148515	0.003190
e250.2	13.592007	0.000000	9.636974	0.000000	6.956299	0.000000	4.864401	0.000020
e250.3	12.287793	0.000000	11.629183	0.000000	9.611706	0.000000	4.865150	0.000020
e250.4	9.984873	0.000000	10.634776	0.000000	8.102196	0.000000	5.961882	0.000001
e250.5	11.158941	0.000000	14.967060	0.000000	9.693668	0.000000	5.257341	0.000006
e250.6	12.340264	0.000000	10.583677	0.000000	7.780299	0.000000	4.895672	0.000018
e250.7	9.009311	0.000000	13.531039	0.000000	7.401949	0.000000	3.231070	0.002548
r100.1	4.877648	0.000009	7.973176	0.000000	3.542336	0.000791	–*	–
r100.2	8.075586	0.000000	14.877051	0.000000	3.212207	0.002151	–*	–
r100.3	6.849298	0.000000	17.023809	0.000000	2.974192	0.004276	–*	–
r100.4	10.412152	0.000000	9.706999	0.000000	4.051202	0.000153	6.940354	0.000000
r100.5	6.905385	0.000000	9.716950	0.000000	5.362519	0.000001	–*	–
r100.6	7.182057	0.000000	11.702706	0.000000	–*	–	–*	–
r100.7	5.388526	0.000001	10.896709	0.000000	5.477226	0.000001	–*	–
r300.1	22.703956	0.000000	16.974441	0.000000	5.903844	0.000000	4.006303	0.000277
r300.2	25.229754	0.000000	26.188688	0.000000	5.968266	0.000000	3.685036	0.000710
r300.3	23.991996	0.000000	24.635767	0.000000	7.823712	0.000000	5.852407	0.000001
r300.4	22.351828	0.000000	22.076844	0.000000	5.610980	0.000001	7.710845	0.000000
r300.5	22.367980	0.000000	26.614378	0.000000	5.569851	0.000001	7.231663	0.000000
r300.6	32.688417	0.000000	26.887646	0.000000	13.870630	0.000000	15.350122	0.000000
r300.7	22.008168	0.000000	23.451034	0.000000	6.599887	0.000000	7.241159	0.000000

* *t* test cannot analyze perfect data

each instance, Julstrom (2005) reported only the typical execution times on instances of each size and type for ESCGA, BCGA and SHC. Moreover, ESCGA, BCGA and SHC were executed on a Pentium 4 processor with 256 MB RAM, running at 2.53 GHz under Red Hat Linux 9.0, which is different from the one used to execute ABC + LS and PABC. Because of these two reasons, it is not possible to exactly compare the execution time of ABC + LS and PABC with those of ESCGA, BCGA and SHC. However, it can be safely concluded that ABC + LS and PABC are faster than BCGA, ESCGA and SHC in instances of size 50 and 100, whereas in instances of size 250 and 300, PABC and ABC + LS are slower than BCGA and SHC but faster than ESCGA. PB-LS (Singh 2008) was executed on the same system as ABC + LS and PABC. ABC + LS and PABC are faster than PB-LS on all instances.

Table 4 Average execution times for ESCGA, BCGA, SHC, PB-LS, PABC and ABC + LS

Instance	Execution time (s)					
	ESCGA[a]	BCGA[a]	SHC[a]	PB-LS	PABC	ABC + LS
e50.1	–	–	–	7.5	4.7	4.7
e50.2	–	–	–	7.9	3.6	3.6
e50.3	–	–	–	7.6	4.4	4.4
e50.4	–	–	–	8.2	3.2	3.2
e50.5	–	–	–	8.7	3.2	3.2
e50.6	–	–	–	8.2	4.1	4.1
e50.7	–	–	–	8.0	4.1	4.1
Typical time	21.2	9.2	9.6	–	–	–
e100.1	–	–	–	54.7	29.5	29.7
e100.2	–	–	–	53.9	28.7	28.9
e100.3	–	–	–	60.6	25.0	25.1
e100.4	–	–	–	53.5	24.8	25.1
e100.5	–	–	–	50.5	29.3	29.5
e100.6	–	–	–	56.5	28.6	28.9
e100.7	–	–	–	55.4	28.3	28.5
Typical time	91.3	36.6	40.0	–	–	–
e250.1	–	–	–	590.5	253.3	266.9
e250.2	–	–	–	573.3	265.6	277.2
e250.3	–	–	–	590.7	291.1	303.0
e250.4	–	–	–	573.0	296.5	309.5
e250.5	–	–	–	563.3	283.9	296.5
e250.6	–	–	–	605.0	363.8	377.0
e250.7	–	–	–	585.4	307.7	321.0
Typical time	618.2	231.1	258.4	–	–	–
r100.1	–	–	–	52.9	16.1	16.3
r100.2	–	–	–	54.5	16.2	16.3
r100.3	–	–	–	52.6	11.0	11.1
r100.4	–	–	–	51.6	16.4	16.5
r100.5	–	–	–	54.9	16.3	16.5
r100.6	–	–	–	54.4	15.9	16.0
r100.7	–	–	–	53.1	14.6	14.8
Typical time	92.6	37.4	40.1	–	–	–
r300.1	–	–	–	709.1	465.1	472.4
r300.2	–	–	–	674.3	302.0	307.9
r300.3	–	–	–	697.1	460.3	467.8
r300.4	–	–	–	668.2	359.0	364.7
r300.5	–	–	–	681.1	267.8	272.6
r300.6	–	–	–	681.3	587.9	600.0
r300.7	–	–	–	689.4	377.9	383.5
Typical time	905.5	351.7	377.8	–	–	–

[a] Execution time on Pentium 4, 2.53 GHz

5 Conclusions

In this paper we have proposed a new approach ABC + LS combining an artificial bee colony (ABC) algorithm with a local search heuristic for the MRCST problem and compared it with best heuristic approaches reported in the literature. On the benchmark instances considered our approach outperformed other approaches in terms of solution quality. Except for one previously proposed approach, our approach is faster than other approaches. We have also shown that ABC algorithm without local search is able to outperform the existing heuristics, thereby, establishing the ABC algorithm to be an effective approach for the MRCST problem.

Acknowledgments We thank Prof. Bryant A. Julstrom for providing the random data set. We also thank two anonymous reviewers for their valuable comments and suggestions which helped in improving the content of this paper. We are grateful to the Department of Science and Technology, Government of India, for their financial support to carry out this research work.

References

Aho AV, Hopcroft JE, Ullman JD (1983) Data structures and algorithms. Addison-Wesley, Reading, pp 232–233

Basturk B, Karaboga D (2006) An artificial bee colony (ABC) algorithm for numeric function optimization, IEEE Swarm Intelligence Symposium 2006, May 12–14, 2006. Indianapolis, Indiana, USA

Campos R, Ricardo M (2008) A fast algorithm for computing the minimum routing cost spanning tree. Comput Netw 52:3229–3247

Fischetti M, Lancia G, Serafini P (2002) Exact algorithms for minimum routing cost trees. Networks 39:161–173

Grout V (2005) Principles of cost minimization in wireless networks. J Heuristics 11:115–133

Johnson DS, Lenstra JK, Roonoy Kan AHG (1978) The complexity of network design problem. Networks 8:279–285

Julstrom BA (2002) A genetic algorithm and two hill climbers for the minimum routing cost spanning tree problem. In: Arabnia HR, Mun Y (eds) Proceedings of the international conference on artificial intelligence, vol III, CSREA Press, pp 934–940

Julstrom BA (2005) The Blob code is competitive with edge-sets in genetic algorithms for the minimum routing cost spanning tree problem. In: Beyer HG et al (eds) Proceedings of the genetic and evolutionary computation conference 2005 (GECCO-2005), vol 1, pp 585–590, ACM Press

Karaboga D (2005) An idea based on honey bee swarm for numerical optimization, Technical Report TR06. Computer Engineering Department, Erciyes University, Turkey

Karaboga D, Basturk B (2007a) A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. J Glob Optim 39:459–471

Karaboga D, Basturk B (2007b) Artificial bee colony (ABC) optimization algorithm for solving constrained optimization problems. Lecture Notes in Artificial Intelligence, Springer, Berlin, vol 4529, pp 789–798

Karaboga D, Basturk B (2008) On the performance of artificial bee colony (ABC) algorithm. Appl Soft Comput 8:687–697

Karaboga D, Basturk B (2009) A comparative study of artificial bee colony algorithm. Appl Math Comput 214:108–132

Picciotto S (1999) How to encode a tree. PhD thesis, University of California, San Diego

Prim RC (1957) Shortest connection networks and some generalizations. Bell Syst Tech J 36:1389–1401

Raidl GR, Julstrom BA (2003) Edge-sets: an effective evolutionary coding of spanning trees. IEEE Trans Evol Comput 7:225–239

Singh A (2008) A new heuristic for the minimum routing cost spanning tree problem. In: Proceedings of the 11th international conference on information technology (ICIT-2008), 9–13, IEEE CS Press

Singh A (2009) An artificial bee colony algorithm for the leaf-constrained minimum spanning tree problem. Appl Soft Comput 9:625–631

Wong R (1980) Worst case analysis of network design problem heuristics SIAM. J Algebraic Discrete Math 1:51–63

Wu BY, Chao KM (2004) Spanning trees and optimization problems, chapter 4. CRC Press, New York

Wu BY, Lancia G, Bafna V, Chao KM, Ravi R, Tang CY (1999) A polynomial time approximation schemes for minimum routing cost spanning trees. SIAM J Comput 29:761–768