Development and Experimental Comparison of Exact and Heuristic Algorithms for the Maximum-Leaf Spanning Tree Problem

Technical Report TR-09-08

Christian Brüggemann[*], Tomasz Radzik

Department of Computer Science, King's College London

Abstract

Given a connected, undirected and unweighted graph G, the maximum leaf spanning tree problem (MLSTP) is to find the spanning tree of G, that has the maximum number of leaves. In this report, we present experimental evaluation of performance of exact algorithms for this problem and several heuristic solutions. The heuristics which we have evaluated include constructive heuristics and heuristics based on Simulated Annealing (SA), Genetic Algorithms (GA) and the recently proposed Artificial Bee Colony method (ABC)[1]. We observe that the ABC algorithm can offer better performance than the SA algorithm, but in most cases the GA algorithm gives the best solutions.

1 Introduction

Given a connected, undirected and unweighted graph G, we consider the maximum leaf spanning tree problem (MLSTP) as to find a spanning tree of G with a maximum number of leaves. The MLSTP is known to be NP-complete [7] and MAX SNP-complete [8]. Finding the maximum leaf spanning tree of a graph has many applications, for instance in circuit layouts. In this report, we present experimental evaluation of performance of exact algorithms for this problem and several heuristic solutions. The heuristics which we have evaluated include constructive heuristics and heuristics based on Simulated Annealing (SA), Genetic Algorithms (GA) and the recently proposed Artificial Bee Colony method (ABC) [1]. For our experimental evaluations, we apply the algorithms on random graphs as well as complete grid graphs in order to be able to compare our results with those of [4]. In addition to that, we also use incomplete grid graphs. The

[*]Supported by EPSRC Vacation Bursary Scheme for undergraduate students

purpose of this report is to discuss the MLSTP from three different point of views (exact algorithms, heuristic solutions and solutions based on meta-heuristics).

This report is structured in the following way: In Section 2, we describe an exact algorithm for the MLSTP proposed by Fujie [4] and discuss possible changes to this algorithm. Section 3 describes the test instances we generated for our simulations and in Section 4, we present the results obtained from exact algorithms for the MLSTP. We propose a new heuristic approach called Priority-BFS in Section 5 and compare its performance with other known heuristics. In Section 6, we present three different meta-heuristic approaches to the problem, based on Simulated Annealing (SA), Genetic Algorithms (GA) and the Artificial Bee Colony (ABC) algorithm. Finally, in Section 7, we present and discuss the test results obtained from all algorithms.

One of the aims of this work was to investigate the applicability of the relatively new artificial bee colony algorithm approach and to compare it with the well established simulated annealing and genetic algorithm approaches. To summarize our findings, we can say that the artificial bee colony algorithm is applicable to the MLSTP. For some instances of graphs, it gives better results than a simulated annealing algorithm using the same notion of neighbourhood and fitness. On the other hand, for some large sparse graphs (incomplete grids), our SA implementation gives better solutions than the ABC implementation. This indicates the merit of (occasionally) accepting a worse neighbouring configuration, as in the SA approach but not in the ABC approach. However, the genetic algorithm clearly gives the best results for larger instances of graphs (especially for grid graphs).

The Priority-BFS heuristic, which we propose in Section 5, improves on previous constructive heuristics for grid graphs and also gives very good results for random graphs and d-regular graphs. We suspect that the Priority-BFS algorithm can be further improved and more importantly we believe that it can be used in the context of exact algorithms to improve their running time.

2 Exact algorithms for the MLSTP

Fujie [4] proposed an exact algorithm for the MLSTP, which is based on the Branch-and-Bound approach and incorporates a very efficient heuristic for selecting subproblems for expansion. A subproblem is denoted as (S_1, S_0, F), where (S_1, S_0, F) is a partition of V. S_1 is the set of vertices that all have to be leaves. S_0 is the set of vertices that all have to be non-leaves and F is the set of the remaining vertices. A vertex in F can either be a leaf or a non-leaf. Fujie [4] shows that the solution to the following problem (P1) is an upper bound on the number of leaves in any spanning tree extending (S_1, S_0, F).

(P1)

$$\begin{aligned}
\text{maximize} \quad & \sum_{i \in F} \frac{|\delta(i)|}{|\delta(i)|-1} - \sum_{e \in E} d_e x_e + |S_1| \\
\text{subject to} \quad & \\
& x \in ST_G, \\
& x(\delta(i)) \leq 1 \quad (i \in S_1),
\end{aligned}$$

where

$$d_e = \begin{cases} \frac{1}{|\delta(i)|-1} + \frac{1}{|\delta(j)-1|}, & \text{for } e = \{i,j\} \text{ with } i,j \in F \\ \frac{1}{|\delta(i)|-1}, & \text{for } e = \{i,j\} \text{ with } i \in F, j \notin F \\ 0, & \text{otherwise.} \end{cases}$$

$\delta(i)$ is the set of edges adjacent to vertex i

$x = (x_e | e \in E)$ is a vector of edges and $x(\delta(i)) = \sum_{e \in \delta(i)} x_e$ for $i \in V$

ST_G is the set of all spanning trees of G

The initial configuration of the algorithm is $(\emptyset, \emptyset, V)$. The generated configurations are kept on a stack S. Therefore the subproblem selection corresponds to the depth-first search. The branch and bound algorithm works as follows.

1. **(Initialization)** Run three different heuristics:
   - A breadth first search algorithm rooted at every vertex $v \in V$ of the input graph G.
   - The 3-approximation algorithm of Lu and Ravi [5].
   - The 2-approximation algorithm of Solis-Oba [6].

   Let LB be the number of leaves of the best solution of the three algorithms. If $LB = |V| - 1$ then stop. (In that case, the solution obtained is already optimal). Otherwise, push the initial configuration $(\emptyset, \emptyset, V)$ onto the stack S.

2. **(Subproblem selection)** If $S = \emptyset$ then stop. Otherwise, pop (S_1, S_0, F) from S. If $|S_1| + |F| \leq LB$ then go to 2 (no spanning tree consistant with this partial solution can improve LB).

3. **(Checking feasibility)** Check whether there exists a spanning tree of G, where each node in S_1 is a leaf. If not, go to 2. (Remark: One could also check here whether there is a spanning tree such that the set of leaves contains S_1 and the set of non-leaves contains S_0. The latter condition is more difficult to check, but might give a better performance.)

4. **(Updating a lower bound)** If $S_1 > LB$ then, for $v \in S_0 \cup F$, run the breadth first search algorithm rooted at v in the graph $G \backslash S_1$, to make a spanning tree in T in G, of which S_1 is a subset of leaves. Let LB be the number of leaves in T - the improved solution value.

5. **(Upper bounding)** Solve Problem (P1) to compute an upper bound UB on the number of leaves in any spanning tree extending this partial solution (S_1, S_0, F). If $\lfloor UB \rfloor \leq LB$, go to 2 ($\lfloor UB \rfloor$ is the greatest integer not greater than UB).

6. **(Subproblem selection)** Choose $v = argmax\{|\delta_G(u)| : u \in F\}$. Push (G, S_1', S_0, F') followed by (G, S_1, S_0', F') on the stack S, where $S_1' = S_1 \cup v$, $S_0' = S_0 \cup v$ and $F' = F \backslash v$. Go to 2.

In step 5 of the Fujie algorithm, solving Problem (P1) can be done by a standard minimum spanning tree computation in graph $G \backslash S_1$ with edge costs d_e, and then, for $i \in S_1$, connecting i and a vertex $j \in V \backslash S_1$ with the minimum cost $d_{\{i,j\}}$. In step 6 the next subproblem to be considered is the subproblem in which the vertex $v \in F$ with maximum degree is considered as the next element of S_0. This generally is a good heuristic especially for dense graphs. However, when it comes to sparse graphs or grid graphs (especially incomplete grid graphs), this heuristic does not perform well. This is due to the fact that for grid graphs and sparse graphs the degree of most vertices is low. Therefore the heuristic selects more or less arbitrary vertices and the search tree becomes big.

The Fujie algorithm selects the next partial configuration in depth-first order in the search tree of the partial solutions. We investigate whether this can be improved by keeping the generated partial configurations in a priority queue and selecting the one with the highest upper bound. We denote this algorithm as "PQ-Fujie". The PQ-Fujie algorithm has a similar structure to the Fujie algorithm. Instead of a stack S we use a priority queue Q. In step 2, we extract the element with the maximum key from Q, where the key is the upper bound of this subproblem. In step 6, when the two subproblems are generated, we check if they are feasible and calculate their upper bounds (as in step 5 of the Fujie algorithm). Therefore, steps 3 and 5 can be omitted in the PQ-Fujie algorithm.

3 Test graphs

In our experiments we use the following types of graphs. Random graphs and grid graphs were also used in the previous study [4], so we can compare our results with those from [4].

Random graphs. We generate a random graph using two input parameters n and p, where n is the number of vertices in the graph and p is the density of the graph. More precisely, p is the probability of two vertices forming an edge in the graph. If a generated graph is disconnected, we dismiss this graph and generate a new one until we obtain a connected graph.

Random d-regular graphs. We decided to also run experiments on d-regular graphs, that is graphs such that all vertices have degree d. Generating such graphs can be done in the following way. Take d copies of each vertex to create a sequence of vertices of length $n * d$. Randomly permute this sequence. For each node at an odd position in this permuted sequence, put an edge between this node and the next node in the permuted sequence. If the neighbour of a vertex is the same vertex in the graph, we dismiss the graph and generate a new one.

Complete grid graphs. Because random graphs are not very likely to occur in applications of the MLSTP, we decided to focus our evaluation of heuristic and meta-heuristic solutions on grid graphs. Let n denote the number of rows in the grid and m the number of columns. Therefore the number of vertices in the graph is $n * m$. The value (1) given below is the lower bound on the maximum number of leaves in a spanning tree of the grid graph, assuming $n \geq m$; see the construction of spanning trees in grid graphs given in [4].

$$mn - 2n - (m-4) - \left\lfloor \frac{m-4}{3} \right\rfloor (n-2). \tag{1}$$

It is also shown in [4] that the upper bound on the number of leaves is

$$\frac{2}{3}mn. \tag{2}$$

The lower and upper bounds given in (1) and (2) are relatively close, as can be seen in Table 7. Suppose that $m = 3k + 1$ for some $k \geq 1$. Then the upper bound is equal to $2kn + \frac{2}{3}n$, while the lower bound is equal to $2kn - k + 1$. Therefore, the difference between the lower and upper bounds is only $O(n)$.

Incomplete grid graphs. The grid graphs are interesting because they turn out to be significantly harder instances for MLSTP algorithms than random graphs. However, very good constructive solutions for grid graphs are known (see the bounds (1) and (2) and the construction given in [4]), and heuristic approaches have problems reaching even the lower bound. Moreover, we would not be very surprised if someone came up with a construction of a spanning tree of a grid with the maximum possible number of leaves. This motivates experiments with incomplete grid graphs. An incomplete grid graph is a graph with n rows and m columns, that has k edges missing. We generate such graphs by removing k random edges from an $n * m$ grid graph. If the graph becomes disconnected, we dismiss it and run the same process again until we obtain a graph that is connected. Incomplete grid graphs may actually reflect better inputs in some applications of the MLSTP than complete grid graphs.

4 Results of exact algorithms for the MLSTP

Tables 2, 3 and 5 compare the performance of the Fujie and PQ-Fujie algorithms. In these tables, the first two columns specify the graphs, columns BFS, PBFS and maxPBFS show the result obtained from those three heuristic approaches (this is discussed later in Section 5) and column opt gives the number of leaves for the maximum leaf spanning tree. Columns Fujie and PQ-Fujie show the performance of our implementations of these algorithms. Columns "problems" shows the number of generated subproblems, until the otimal solution is found, while columns "time" shows the time it took the algorithm to compute the optimum value in seconds. The lower values are shown in bold. Both algorithms have been tested on the same graphs. Each experiment has been done 20 times and the average values are given in the tables.

It turns out that for random graphs the Fujie algorithm which uses a priority queue (algorithm PQ-Fujie) gives better results only if the graph is relatively sparse (see Table 2). For dense graphs the heuristic of selecting nodes with many edges as interior nodes seems to be a much better mechanism than using a priority queue. Table 3 shows, that the Fujie algorithm is about 3 times faster than the PQ-Fujie algorithm on d-regular graphs. However, we could only evaluate a very limited number of experiments, since the computation on larger graphs was not feasible.

Table 5 shows, that the implementation using a priority queue clearly outperformes the original Fujie algorithm on grid graphs with randomly removed edges. As mentioned earlier, this might be due to the fact that many vertices in set F have the same degree.

5 Constructive heuristics for the MLSTP

The Breadth-First-Search traversal of graph G gives relatively good solutions to the MLSTP for random graphs and random d-regular graphs. However, for grid graphs it gives very poor solutions, as can be seen in Table 1 (see also Tables 6 and 7 for results on larger grids). For instance on a $20 * 40$ complete grid graph the breadth first search gives a tree with only 80 leaves, although the lower bound (1) is 514. The approximation algorithm proposed by Lu and Ravi [5] as well as the approximation algorithm proposed by Solis-Oba [6] both offer a significant improvement compared to the solution obtained from BFS on grid graphs.

We would like to have even better constructive heuristics to be able to compute fast good solutions. Such heuristics could be used within an exact algorithm to provide good initial lower bounds. More importantly, we need good constructive heuristics to compute starting configurations for meta heuristics.

We propose a heuristic which we call Priority-BFS, and show that it outperformes BFS as well as Lu-Ravi and Solis-Oba on grid graphs. We use this heuristic to generate initial solutions in the metaheuristic algorithms discussed in Section 6.

In Table 1 we compare the new heuristic against the standard breadth first search and the algorithms proposed by Lu-Ravi and Solis-Oba. The BFS algorithm is running the breadth first search from every vertex $v \in V$ and returning the best result. In this report, BFS algorithm will always denote this procedure. The data for Lu-Ravi and Solis-Oba shown in Table 1 are quoted from [4] (we did not implement those algorithms). Max-Priority-BFS is the Priority-BFS algorithm started from all vertices of G and then the best result is selected. This version of Priority-BFS runs in $O(n^2)$ time.

The pseudocode for algorithm Priority-BFS is given below. This algorithm maintains two sets T (the tree built so far) and Q (the set of leaves of the tree). Initially T is empty and Q contains a vertex with maximum degree. During the main while-loop the following invariant holds: T is the set of edges already in the final tree and Q is the set of leaves of T. At each iteration node i in Q with the maximum number of neighbours outside T is selected and deleted from Q. All neighbours of node i outside of T are added to Q and T. At the end of the computation T is the final tree and Q is empty. It is very efficient to implement Q as a priority queue, where the key of a vertex is equal to $degree_G(v) - degree_T(v)$.

Algorithm 1 Priority-BFS

```
1: function PRIORITY-BFS(Graph $G$)                                   ▷ The MLST of G
2:     $start \leftarrow$ a vertex with maximum degree
3:     $T \leftarrow \emptyset$                                               ▷ $T$ is the built tree
4:     $Q \leftarrow \emptyset$                        ▷ empty priority queue with the leaves in $T$
5:     $push(Q, (start, degree(start)))$
6:     while $\neg empty(Q)$ do
7:         $i \leftarrow extract\_max(Q)$        ▷ The vertex with max $degree_G(v) - degree_T(v)$
8:         for all vertices $v$ adjacent to $i$ do
9:             $enqueue(Q, (v, degree_G(v) - degree_T(v)))$
10:            $T \leftarrow T \cup (i, v)$
11:        end for
12:    end while
13:    return $T$
14: end function
```

Table 1 shows that the performance of Priority-BFS is better than the other three algorithms for grid graphs of size $6 * 7$ and larger. We also observe that in many cases Priority-BFS finds the optimal solution.

Graph								
n	m	BFS	Lu-Ravi	Solis-Oba	Priority-BFS	Max-Priority-BFS	lower bound	optimum
3	3	***6**	***6**	***6**	***6**	***6**	4	6
	4	7	***8**	***8**	***8**	***8**	6	8
	5	9	***10**	***10**	***10**	***10**	8	10
	6	11	***12**	***12**	***12**	***12**	10	12
	7	13	***14**	***14**	***14**	***14**	11	14
	8	15	***16**	***16**	***16**	***16**	13	16
	9	17	***18**	***18**	***18**	***18**	15	18
4	4	8	8	8	8	***9**	8	9
	5	10	10	10	10	***11**	11	11
	6	12	***14**	***14**	13	13	14	14
	7	14	15	***16**	14	15	15	16
	8	16	***18**	***18**	17	***18**	18	18
	9	18	19	***21**	19	20	21	21
5	5	12	13	13	13	***14**	14	14
	6	14	**17**	16	**17**	**17**	18	18
	7	16	19	18	19	***20**	20	20
	8	18	21	20	21	***23**	23	23
	9	20	25	23	25	**26**	27	27
6	6	16	**21**	19	18	**21**	22	22
	7	18	24	22	***26**	***26**	26	26
	8	20	27	25	***30**	***30**	30	30
	9	22	30	28	32	**33**	34	34
7	7	20	27	26	27	***29**	27	29
	8	22	31	30	***33**	***33**	33	33
	9	24	35	34	**37**	**37**	39	39
8	8	24	35	34	***38**	***38**	38	38
	9	26	40	39	**42**	**42**	45	45
9	9	28	45	43	47	**48**	51	51

Table 1: *comparing Priority-BFS to Solis-Oba, Lu-Ravi and BFS on complete grid graphs. The * symbol indicates that the value is otimal.*

6 Solutions for MLSTP based on meta-heuristics

We have developed several heuristic solutions for the MLSTP based on meta-heuristic approaches: a simulated annealing algorithm (SA), a genetic algorithm (GA) and an artificial bee colony algorithm (ABC). In the following subsections we discuss our three implementations and also the test results. We mainly focused our experiments and implementations on complete grid graphs as well as grid graphs with randomly removed edges, because it is harder to solve the MLSTP on such graphs. For random graphs, a simple breadth first search gives nearly optimal solutions to the problem, whereas on grid graphs it gives extremely poor solutions.

6.1 Simulated Annealing

Our SA algorithm uses the tree obtained from the Priority-BFS heuristic as the starting configuration. For a tree T, a random neighbouring tree T' is obtained in the following standard way.

1. Remove a random edge e_{ij} from the tree T, giving two disconnected components.
2. Find a random edge $(e_{i'j'} | i \neq i' \vee j \neq j')$ that connects the two components again.
3. Add $e_{i'j'}$ to the tree in order to obtain a new tree T'.

The cost function in the MLSTP counts only the number of leaves in the tree. Using the neighbourhood we propose, it will only be possible to find a neighbour with more leaves by deleting an edge that has at least one vertex of degree 2. Therefore in our fitness function, we give a small weight also to degree 2 vertices which then could possibly be transformed to leaves in the next iteration. This helps to distinguish between the fitness of two trees with the same number of leaves. We use the following fitness function:

$$fit(T) = \sum_{i=0}^{n} \begin{cases} 1, & \text{if } degree(i) = 1, \\ \frac{1}{n}, & \text{if } degree(i) = 2, \\ \frac{0.5}{n}, & \text{if } degree(i) = 3, \\ 0, & \text{if } degree(i) > 3. \end{cases} \tag{3}$$

Equation (3) has been designed especially for grid graphs (or more generally for low degree graphs). For other type of graphs, one should consider giving some positive weights also to nodes with degrees > 3. An important issue in the simulated annealing algorithm is the probability of changing to a worse configuration, which in our implementation is defined as

$$p = e^{\frac{a-x-1}{c}}, \tag{4}$$

where a is the fitness of the new configuration, x is the fitness of the current configuration and c is the current temperature. We decided to subtract 1 from $a - x$ in Equation (4) in order not to allow the algorithm to always move to a configuration with the same fitness. Our preliminary experiments showed that allowing the algorithm to change to configurations with the same fitness led to worse results. Initially, we set the temperature c to 1 and at each iteration update $c = c * 0.995$. We decided to make c independet of the size of the graph, because $a - x - 1$ is independend of the size of the graph. This is because the difference between the number of leaves in the current tree and a neighbouring tree is between -2 and 2. The values 1 and 0.995 were obtained from experiments.

6.2 Genetic Algorithm

Our genetic algorithm solution has been implemented in the following way. The population consits of N individuals. The mutation operation is the same as the neighboring function in the simulated annealing algorithm and occurs with probability p_m on all individuals after the cross-over phase. On the initialization, each individual is assigned a tree obtained by the Priority-BFS heuristic using a random starting vertex. In that way, the initial individuals are good solutions and very likely to be different from one another. At each iteration of the algorithm, the population is sorted by fitness. The fitness of an individual is defined by Equation (3). The fittest $\lceil \frac{N}{2} \rceil$ individuals are kept in the next generation, and the rest of the individuals are abandoned. Instead, $\lfloor \frac{N}{2} \rfloor$ new individuals will be generated through cross-over. For the cross-over operation each individual in the fitter half gets paired to a random individual in the same half. The operation cross-over for the parent trees A and B works in the following way:

1. Take together the edges of tree A and tree B to form a subgraph G'.
2. Use the Priority-BFS algorithm to create a leafy spanning tree of G'. This tree is the created child.

It turns out that this cross-over operation often creates a child which is the same as one of the parents. Allowing such a child to enter the population leads to the situation that after a few iterations, most individuals in the population are the same and the algorithm stops progressing. Therefore we did not allow for creating clones. If it occurs that two parents create a clone, we simply abandon the child and insert the next fittest individual from the lower half of the old population instead.

Our experiments showed that a population size of $N = 250$ and a mutation ratio of $p_m = 0.1$ gives good results for the graphs we tested. In the implementation, each individual in the population, a tree, is represented as an array of parent pointers that form this tree.

It is often the case, and this also happens with our implementation, that GA solutions are time consuming. We note that the cross-over operations of our GA (which take the most computation time) can run in parallel, so improved performance could be achieved on computers with more than one CPU.

6.3 Artificial Bee Colony Algorithm

The Artificial Bee Colony algorithm (ABC) proposed recently by Karaboga and Basturk[1] is a meta-heuristic approach designed for numerical optimization of multivariate functions. It uses terms and notions related to the behaviour of honey bees. The ABC algorithm maintains a "colony" (set) of artificial bees. Each bee is a computational process having one solution associated with it. There are three different groups of bees: Employed bees, onlooker bees and scouts. Let the whole colony consist of N bees, then there are $\frac{N}{2}$ employed bees and $\frac{N}{2}$ onlooker bees. Ocasionally, an employed bee transforms into a scout and then back into an employed bee.

At the initialization of the algorithm, each employed bee gets assigned a random "food source". A food source is a solution to the given problem. Each onlooker bee is not working and there are no scouts. In general, onlooker bees are helping to exploit promising "food sources" found by employed bees. Scouts are employed bees that abandoned the current search (because they cannot find a better solution) and generate a new solution (in our case using the Priority-BFS algorithm from a random start vertex). After obtaining a new "food source" (solution), they become employed again and start to work on that solution. Below are given the main steps of the ABC algorithm adapted from numerical optimisation to discrete optimisation.

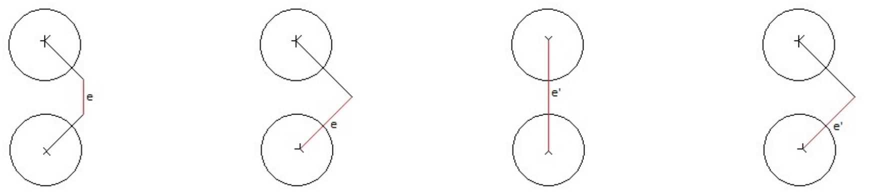
Algorithm 2 Artificial Bee Colony Algorithm

```
procedure ABC
    Initialize: a set of $\frac{N}{2}$ employed bees and $\frac{N}{2}$ onlooker bees;
    each employed bee generates its initial solution; each onlooker bee selects its
    initial solution as a copy of the solution from one of the employed bees selected
    randomly according to probabilities (5)
    while termination conditions are not met do
        for all Bees $B$ do
            Find a neighbouring configuration $C'$ from $B$'s current configuration $C$
            if $C'$ is better than $C$ then
                $B$ moves to $C'$ and forgets $C$
            end if
            if $B$ has not found a better configuration for $MaxCycle$ times then
                if $B$ is an employed bee then
                    $B$ becomes a "scout" to generate a new random starting
                    configuration and then turns back into an employed bee
                else                                        ▷ $B$ is an onlooker bee
                    $B$ chooses an employed bee using Equation (5) and copies its
                    configuration
                end if
            end if
        end for
    end while
end procedure
```

For our Artificial Bee Colony algorithm approach, we have used the same neighboring configuration as in the Simulated Annealing approach but with a different procedure of generating a neighbour. Because an ABC algorithm never accepts a worse configuration, we have designed a neighbouring function that can only return neighbours that have a higher fitness. Consider a spanning tree T of a graph G. The only possibility of obtaining a new spanning tree T' with more leaves than T by deleting only one edge e and inserting a new edge e', where $e \in T$ and $e' \in \{G - T\}$, is that the following conditions on e and e' hold:

1. e has to have at least one vertex of degree 2.
2. e' has to have one vertex of degree > 1, if e has two vertices of degree 2.
3. e' has to have two vertices of degree > 1, if e has precisely one vertex of degree 2.

There are two possibilities of deleting an edge e shown in Figure (1)(a) and (b). We either gain two new leaves or one new leave. For inserting a new edge e' we also have two different possibilities shown in Figure (1) (c) and (d) taking either one leave away or not changing the number of leaves. The algorithm uses the observations made in Figure (1) in order to only produce neighbours with more leaves.



(a) +2 leaves (b) +1 leave (c) +-0 leaves (d) -1 leave

Figure 1: Finding a neighbouring configuration for the ABC algorithm

During each iteration, an onlooker bee that is not working at the moment will be assigned to a food source using probability

$$p_i = \frac{fit_i}{\sum_{j=0}^{SN} fit_j}, \tag{5}$$

where fit_i is the fitness of food source i, SN is the number of employed bees and p_i is the probability with which the onlooker bee will select the food source i.

At each iteration, every bee (including onlooker and employed bees) will generate a neighboring configuration using the method described above. If it can find a better configuration then it will move to the new configuration. If a bee cannot find a better configuration, then it will abandon the food source and either look for another food source of an employed bee (if it is an onlooker bee) or become a scout (if it is an employed bee). A scout will randomly generate a new food source (using the Priority-BFS heuristic with a random start vertex) and then becomes an employed bee again.

Observe that since we generate only a better neighbouring configuration (if any exists), then we do not need to check the conditions in Lines 9 and 12 of Algorithm 2.

7 Test Results

Tables 2 to 7 show the performance of all algorithms we implemented. All algorithms were run on exactly the same graphs and each table shows the performance on a different type of graph. For each graph, we ran 20 experiments. The Fujie and Fujie-PQ algorithms were tested only on small graphs. The column "iter." for the meta-heuristic solutions denotes the number of iterations that it took the algorithm, until it found its best solution. However, the column "time" for the meta-heuristic solutions denotes the total time, that the algorithm took (including all iterations until the stopping conditions were met).

Random graphs. Table 2 shows the experiments for all implemented algorithms on random graphs with 30 to 50 nodes and density from 0.1 to 0.5. For dense as well as for sparse graphs the breadth first search achieves a relatively good solution. The priority-BFS (PBFS) algorithm outperformes the BFS algorithm on sparse random graphs. On dense random graphs (starting with density 0.2), the BFS algorithm achieves better solutions. Although it has been designed for grid graphs, in all our experiments the Max-PBFS algorithm outperformes the BFS as well as the PBFS algorithm. For random graphs the Max-PBFS algorithm gets very close to the optimum solution (on average, there is less than one leaf missing).

The Fujie algorithm is faster than the Fujie algorithm implemented using a priority queue (PQFujie) for dense graphs (usually with density > 0.2). For sparse graphs, the PQFujie algorithm is significantly faster than the Fujie algorithm.

Out of the three meta-heuristic approaches, the genetic algorithm (GA) gives better results than the other two algorithms. However, its running time is 2-5 times higher than that of the simulated annealing (SA) and artificial bee colony (ABC) algorithms. It is interesting, that the GA in most cases finds the optimum solution (even for sparse graphs, where finding the solution is a much harder task). Note also, that the GA as well as the ABC algorithm find their best solution within very few iterations.

***d*-regular graphs.** The BFS algorithm gives better results for d-regular graphs than the Priority-BFS algorithm. However, the MaxPBFS algorithm outperformes the BFS algorithm. We were only able to do limited experiments, because larger graphs take too long for the Fujie and PQFujie algorithms. On those experiments in Table 3, the Fujie algorithm performs better than the PQFujie algorithm. This might be due to the fact that the distance between any two vertices is relatively near, in other words, the graph is too closely connected. Similar than the dense random graphs, here the selection process of the Fujie algorithm is very efficient.

On d-regular graphs, the solutions based on meta-heuristics achieve results very near the optimum. The GA is clearly the best amongst them for this type of graph.

Small complete grid graphs. On small grid graphs, a breadth-first search gives extremely poor results. As Table 4 shows, the BFS algorithm finds only 35% of the number of leaves possible on a 9 by 9 grid. In general, a BFS algorithm gives poor results on any type of sparse or grid graph. However, the PBFS as well as the Max-PBFS algorithms find very good solutions, which are in some cases the optimum solution.

All three meta-heuristic approaches give very good (mostly optimal) solutions for small grid graphs. The GA and the SA algorithm outperform the ABC algorithm on such graphs.

Small incomplete grid graphs. The experiments for small incomplete grid graphs have been made on n x m graphs with k removed edges, where $k = \frac{2nm}{10}, k = 2 * \frac{2nm}{10}, k = 3 * \frac{2nm}{10}$. We denote these three cases as case (1), (2) and (3) respectively. As Table 5 shows, the PBFS algorithm always gives as least as good results as the BFS algorithm. In case (1), the PBFS algorithm gives significantly better results than BFS. Note also, that MaxPBFS gives even better results than BFS for cases (1) and (2).

The PQFujie algorithm finds the optimum solution significantly faster than the Fujie algorithm in all three cases. For some experiments, we could not finish the Fujie algorithm, because it did not find the solution in a reasonable time.

Interestingly, all three meta-heuristic approaches improve on the PBFS algorithm. In most cases, they find the optimum solution in very few iterations.

Large complete grid graphs. The BFS algorithm gives extremely bad results for complete grid graphs from 10 x 10 up to 20 x 40 nodes, as Table 7 shows. However, MaxPBFS achieves results relatively near the lower bound. The lower bound and upper bound are taken from Equations (1) and (2). All three meta-heuristic solutions almost reach the lower bound, but as the graphs grow bigger, the GA clearly outperforms the other two. Interestingly, the performance of the ABC algorithm lies somewhere between the SA and the GA.

Large incomplete grid graphs. We also removed $k = \frac{2nm}{10}, k = 2 * \frac{2nm}{10}$ edges from the n x m grid graphs for large instances. Removing more edges makes it hard to generate such graphs and would not give appropriate test instances. As Table 6 shows, the BFS algorithm gives extremely bad results for large incomplete grid graphs. The Priority-BFS algorithm improves significantly on the BFS and the MaxPBFS algorithm also improves on the PBFS algorithm. The SA and GA still improve on the result of the MaxPBFS algorithm, however the ABC algorithm does not improve very much.

Graph		BFS	PBFS	MaxPBFS	Opt.	Fujie		PQFujie		SA			ABC			GA		
n	p	sol.	sol.	sol.		prob.	time	prob.	time	sol.	iter.	time	sol.	iter.	time	sol.	iter.	time
30	0.1	17.10	17.55	18.60	18.95	19649.9	1.87	189.9	**0.01**	18.85	720.9	0.34	18.85	9.8	0.19	18.85	0.3	1.44
	0.2	21.20	21.60	22.95	23.55	15969.4	1.77	2856.6	**0.24**	23.25	1480.2	0.34	23.20	17.2	0.22	23.50	1.1	1.41
	0.3	23.95	23.85	25.00	25.45	4364.5	0.62	3151.0	**0.28**	25.10	1369.8	0.34	25.15	4.2	0.28	**25.45**	1.0	1.46
	0.4	25.10	24.50	26.00	26.35	1363.9	0.22	1678.9	**0.16**	25.80	1188.6	0.34	26.10	7.7	0.36	26.30	0.4	1.50
	0.5	26.50	25.65	26.95	27.05	373.9	**0.07**	678.2	0.07	26.35	349.5	0.34	27.00	5.1	0.49	**27.05**	0.8	1.54
40	0.1	24.65	25.50	27.15	28.05	940001.3	142.45	3949.2	**0.48**	27.65	2042.4	0.58	27.65	34.0	0.34	27.75	1.2	2.05
	0.2	30.50	30.20	32.15	33.40	31268.5	**5.73**	37893.5	10.74	32.60	2277.6	0.56	32.60	31.9	0.39	33.15	1.6	2.15
	0.3	33.15	32.50	34.30	35.05	18828.6	4.22	19691.5	**4.04**	34.25	2212.5	0.55	34.65	28.2	0.52	**35.05**	1.9	2.27
	0.4	35.05	34.50	35.75	36.20	5733.2	1.60	7029.3	**1.20**	35.40	1105.6	0.55	35.95	11.9	0.73	36.15	0.5	2.28
	0.5	35.95	35.30	36.45	37.00	852.7	**0.28**	3297.9	0.58	36.10	1434.2	0.54	36.45	2.1	0.87	36.85	0.4	2.36
50	0.1	33.00	34.50	36.90	38.10	14895916.4	2904.82	126355.3	**116.07**	37.70	2977.7	0.82	37.40	40.0	0.47	37.20	2.8	3.00
	0.2	39.90	39.75	41.65	43.25	228427.2	**62.76**	343507.3	920.63	42.25	3409.1	0.82	42.40	122.0	0.71	42.90	10.0	3.20
	0.3	42.90	42.10	44.05	44.95	85692.5	**30.85**	137992.6	154.82	43.70	2216.1	0.83	44.30	40.3	0.91	**44.95**	2.0	3.03
	0.4	44.60	43.75	45.10	46.00	15393.0	6.76	17802.2	**5.14**	44.65	1802.3	0.80	45.35	18.7	1.20	45.90	1.3	3.26
	0.5	45.90	45.00	46.35	47.00	1121.5	**0.61**	9049.7	2.68	45.75	1529.5	0.80	46.35	1.2	1.57	46.85	1.6	3.42
60	0.4	54.40	53.50	55.20	56.05	28699.5	20.28	32174.6	**17.15**	54.45	2277.1	1.18	55.30	7.0	2.07	56.00	2.1	4.88
	0.5	55.75	54.70	56.10	56.70	17326.9	**14.88**	30410.5	19.41	55.20	1271.7	1.22	56.10	0.2	2.80	56.45	0.6	5.27
	0.6	56.40	55.45	56.95	57.00	3094.9	3.37	4804.0	**2.76**	55.95	942.6	1.27	**57.00**	13.6	3.87	**57.00**	0.1	5.65

Table 2: *experiments on small random graphs*

Graph		BFS	PBFS	MaxPBFS	Opt.	Fujie		PQFujie		SA			ABC			GA		
n	d	sol.	sol.	sol.		prob.	time	prob.	time	sol.	iter.	time	sol.	iter.	time	sol.	iter.	time
30	3	13.55	13.20	14.95	15.95	6935.7	0.82	6383.0	**0.77**	15.55	2268.3	0.47	15.15	10.4	0.26	15.90	2.1	1.81
	4	17.25	17.15	18.75	20.00	11920.8	1.20	7351.0	**0.71**	19.35	1375.9	0.37	19.20	7.6	0.21	19.95	2.5	1.59
	5	19.45	18.75	20.40	22.00	22756.2	**2.88**	21934.2	6.30	21.40	1396.4	0.44	20.85	23.1	0.25	21.80	2.8	1.90
40	3	17.60	17.40	19.70	21.00	72643.4	**11.59**	50680.7	34.38	20.50	3044.2	0.70	20.10	53.0	0.40	20.95	5.0	2.64
	4	22.20	22.60	24.90	26.40	242353.0	**37.67**	138030.6	129.00	25.60	2859.0	0.65	25.30	12.4	0.35	26.15	3.0	2.63

Table 3: *experiments on small d-regular graphs*

Graph		Opt.	BFS	PBFS	MaxPBFS	SA			ABC			GA		
n	m		solution	solution	solution	solution	iter.	time	solution	iter.	time	solution	iter.	time
4	4	9	8	8	**9**	**9**	14	0.16	**9**	0	0.07	**9**	0	0.68
	5	11	10	10	**11**	**11**	134	0.17	**11**	0	0.09	**11**	0	0.93
	6	14	12	13	13	**14**	198	0.23	**14**	0	0.12	**14**	1	1.12
	7	16	14	14	15	**16**	623	0.33	**16**	39	0.17	**16**	1	1.39
	8	18	16	17	**18**	**18**	177	0.39	**18**	0	0.21	**18**	0	1.72
	9	21	18	19	20	20	88	0.48	20	0	0.23	**21**	1	1.90
5	5	14	10	13	**14**	**14**	28	0.26	**14**	0	0.14	**14**	0	1.23
	6	18	12	17	17	**18**	1246	0.35	17	0	0.17	**18**	1	1.45
	7	20	14	19	**20**	**20**	3	0.45	**20**	0	0.21	**20**	0	1.72
	8	23	16	21	**23**	**23**	1737	0.58	**23**	0	0.31	**23**	0	2.11
	9	27	18	25	**26**	25	0	0.69	**26**	0	0.31	**26**	0	2.38
6	6	22	12	18	21	**22**	999	0.45	**22**	48	0.24	**22**	1	1.85
	7	26	14	**26**	**26**	**26**	0	0.59	**26**	0	0.29	**26**	0	2.19
	8	30	16	**30**	**30**	**30**	0	0.74	**30**	0	0.34	**30**	0	2.56
	9	34	18	32	33	**34**	700	0.90	**34**	1	0.51	**34**	1	3.13
7	7	29	14	27	**29**	**29**	143	0.76	**29**	0	0.37	**29**	0	2.89
	8	33	16	**33**	**33**	**33**	0	0.99	**33**	0	0.47	**33**	0	3.49
	9	39	18	37	37	**39**	3714	1.25	37	0	0.64	38	2	3.85
8	8	38	16	34	36	**38**	5019	1.29	37	11	0.76	**38**	1	4.36
	9	45	18	42	42	**45**	8693	1.62	43	145	0.97	**45**	41	6.13
9	9	51	18	47	48	**51**	4884	1.99	50	149	1.29	**51**	2	5.53

Table 4: *experiments on small complete grid graphs*

Graph			BFS	PBFS	MaxPBFS	Opt.	Fujie		PQ-Fujie		SA			ABC			GA		
n	m	k	solution	solution	solution		subprob.	time	subprob.	time	solution	iter.	time	solution	iter.	time	solution	iter.	time
4	4	3	7.75	7.90	**8.50**	**8.50**	273.5	0.01	36.4	0.00	**8.50**	56.4	0.13	**8.50**	0.4	0.07	**8.50**	0.0	0.60
		6	6.95	6.95	**7.10**	**7.10**	105.5	0.00	5.8	0.00	**7.10**	42.3	0.13	**7.10**	0.9	0.08	**7.10**	0.0	0.61
		9	5.80	**5.80**	**5.80**	**5.80**	18.0	0.00	1.0	0.00	**5.80**	0.0	0.13	**5.80**	0.0	0.09	**5.80**	0.0	0.56
	5	4	9.80	10.00	**10.50**	**10.50**	1038.6	0.05	58.3	0.00	**10.50**	115.9	0.19	**10.50**	0.5	0.11	**10.50**	0.0	0.84
		8	8.75	8.80	**9.10**	**9.10**	276.3	0.01	11.6	0.00	**9.10**	16.5	0.19	**9.10**	0.8	0.12	**9.10**	0.0	0.82
		12	**7.40**	**7.40**	**7.40**	**7.40**	22.0	0.00	1.0	0.00	**7.40**	0.0	0.20	**7.40**	0.0	0.15	**7.40**	0.0	0.77
	6	5	11.45	11.45	12.45	**12.55**	4917.4	0.35	126.9	0.01	12.40	332.4	0.24	12.50	8.4	0.14	**12.55**	0.5	1.05
		10	10.10	10.30	10.40	**10.55**	596.6	0.04	22.8	0.00	**10.55**	86.8	0.25	10.50	0.4	0.17	10.50	0.3	1.07
		15	**8.40**	**8.40**	**8.40**	**8.40**	26.0	0.00	1.0	0.00	**8.40**	0.0	0.25	**8.40**	0.0	0.21	**8.40**	0.0	0.96
	7	6	13.20	13.45	14.40	**14.55**	15063.6	1.24	308.1	0.02	**14.55**	649.8	0.30	14.45	0.1	0.19	**14.55**	0.6	1.33
		12	12.60	12.70	13.05	**13.10**	1198.5	0.09	31.2	0.00	**13.10**	138.9	0.31	**13.10**	3.9	0.21	**13.10**	4.1	1.39
		18	**9.40**	**9.40**	**9.40**	**9.40**	30.0	0.00	1.0	0.00	**9.40**	0.0	0.35	**9.40**	0.0	0.35	**9.40**	0.0	1.37
	8	6	15.05	15.60	16.75	**17.10**	137288.5	13.58	1048.5	0.07	**17.10**	1145.6	0.39	17.05	47.4	0.26	17.05	0.3	1.62
		12	13.85	14.15	14.75	**14.85**	5864.0	0.57	110.3	0.01	**14.85**	200.8	0.40	14.80	0.2	0.29	**14.85**	0.1	1.63
		18	12.60	12.60	**12.65**	**12.65**	196.6	0.01	5.1	0.00	**12.65**	0.35	0.42	**12.65**	1.0	0.36	**12.65**	0.0	1.67
	9	7	17.45	17.80	19.15	**19.25**	369687.5	42.97	1253.0	0.10	**19.25**	785.0	0.48	**19.25**	7.6	0.30	**19.25**	0.1	1.91
		14	14.95	15.60	16.15	**16.25**	39439.0	4.39	240.4	0.02	**16.25**	223.1	0.48	**16.25**	1.9	0.36	**16.25**	0.1	1.97
		21	13.00	13.00	13.25	**13.35**	302.6	0.03	12.7	0.00	**13.35**	20.3	0.51	**13.35**	0.7	0.43	**13.35**	0.1	1.85
5	5	5	11.05	12.10	12.85	**13.00**	12814.7	0.97	288.7	0.01	**13.00**	214.2	0.26	12.95	7.3	0.16	12.95	0.1	1.15
		10	10.55	10.60	**11.25**	**11.25**	1263.7	0.09	42.3	0.00	**11.25**	41.6	0.26	**11.25**	0.5	0.18	**11.25**	0.0	1.15
		15	9.05	9.05	**9.10**	**9.10**	55.35	0.00	2.3	0.00	**9.10**	0.1	0.29	**9.10**	1.0	0.23	**9.10**	0.0	1.09
	6	6	13.20	14.65	15.75	**16.15**	58798.9	5.51	541.2	0.03	16.00	396.1	0.35	16.05	8.9	0.20	16.05	0.4	1.48
		12	12.85	13.00	13.70	**13.95**	6298.5	0.57	80.8	0.00	**13.95**	128.0	0.36	13.80	0.4	0.23	**13.95**	4.1	1.54
		18	10.85	**10.95**	**10.95**	**10.95**	152.9	0.01	4.5	0.00	**10.95**	0.0	0.37	**10.95**	0.0	0.31	**10.95**	0.0	1.44
	7	7	15.90	17.25	18.35	**18.65**	448718.5	49.02	2665.5	0.22	18.35	455.9	0.45	18.55	3.2	0.27	18.60	0.5	1.76
		14	15.20	16.00	16.75	**16.85**	15409.8	1.66	118.5	0.01	**16.85**	208.3	0.46	**16.85**	0.2	0.30	**16.85**	0.2	1.77
		21	12.40	12.65	**12.80**	**12.80**	206.0	0.02	16.9	0.00	**12.80**	1.2	0.48	**12.80**	0.0	0.43	**12.80**	0.0	1.80
	8	8	17.80	19.90	21.10	**21.60**	3046231.0	431.51	5686.0	0.81	**21.60**	1597.3	0.64	21.40	26.5	0.41	21.40	0.7	2.54
		16	16.70	18.40	19.20	**19.40**	41295.5	5.74	302.8	0.03	**19.40**	105.5	0.63	**19.40**	15.0	0.45	**19.40**	0.2	2.41
		24	14.75	15.05	**15.45**	**15.45**	435.4	0.05	39.5	0.00	**15.45**	58.2	0.62	**15.45**	0.6	0.57	**15.45**	0.0	2.41
	9	9	19.90	21.85	23.50	**24.30**	n/A	n/A	22766.8	6.42	23.55	1724.4	0.69	23.95	52.4	0.43	**24.30**	1.4	2.51
		18	18.35	20.25	21.05	**21.40**	203402.0	30.06	731.4	0.07	**21.40**	858.5	0.70	21.30	2.4	0.48	**21.40**	0.5	2.63
		27	17.60	18.00	**18.20**	**18.20**	2032.6	0.29	43.6	0.00	**18.20**	11.4	0.74	**18.20**	0.8	0.62	**18.20**	0.0	2.74
6	6	7	15.20	17.50	19.40	**19.80**	782113.7	104.64	2584.6	0.27	**19.80**	1315.5	0.53	19.70	91.4	0.37	**19.80**	3.5	2.28
		14	14.90	16.30	17.30	**17.50**	30565.8	3.40	152.2	0.01	**17.50**	258.9	0.47	17.40	1.6	0.32	17.40	0.1	1.88
		21	13.90	14.20	**14.50**	**14.50**	612.9	0.07	31.2	0.00	**14.50**	11.1	0.49	**14.50**	0.7	0.38	**14.50**	0.0	1.83
	7	8	17.10	20.45	22.45	**23.15**	n/A	n/A	8830.9	1.26	**23.15**	1666.0	0.61	22.85	22.3	0.39	**23.15**	3.6	2.34
		16	18.10	19.10	20.40	**20.70**	417144.8	55.57	392.5	0.04	**20.70**	507.9	0.62	20.60	20.6	0.42	**20.70**	0.3	2.37
		24	15.35	16.10	**16.65**	**16.65**	5465.7	0.74	67.9	0.01	**16.65**	38.5	0.65	**16.65**	0.5	0.54	**16.65**	0.0	2.54
	8	10	20.15	24.60	26.25	**26.70**	n/A	n/A	49101.9	38.08	26.00	840.2	0.78	26.45	34.8	0.43	**26.70**	0.8	2.66
		20	19.70	21.50	22.40	**22.80**	2114803.7	346.82	985.8	0.13	22.70	336.6	0.92	**22.80**	55.1	0.63	22.70	0.4	3.43
		30	16.85	17.25	**17.90**	**17.90**	4089.0	0.68	66.1	0.01	**17.90**	48.0	0.85	**17.90**	0.5	0.84	**17.90**	0.0	3.22

Table 5: *experiments on small incomplete grid graphs*

Graph			BFS	PBFS	MaxPBFS	SA			ABC			GA		
n	m	k	solution	solution	solution	solution	iter.	time	solution	iter.	time	solution	iter.	time
10	10	20	34.00	50.75	54.30	53.80	2612.2	2.85	54.90	106.6	1.64	**56.50**	3.0	7.63
		40	37.10	45.70	48.65	49.10	2394.3	2.93	49.30	231.2	2.07	**50.30**	2.9	8.69
	15	30	53.95	75.55	80.95	84.85	5971.7	6.27	81.75	246.3	4.17	**85.85**	11.7	16.60
		60	55.45	70.75	74.95	**78.10**	5170.2	6.43	75.85	234.0	4.88	77.95	4.8	19.11
	20	40	71.65	102.35	108.60	112.80	7070.7	10.81	109.05	172.9	6.76	**114.70**	9.0	26.49
		80	76.15	94.60	99.60	104.90	6545.6	10.91	100.65	286.6	8.26	**105.25**	12.2	28.99
	40	80	149.75	201.90	216.05	222.20	8818.2	43.64	216.30	352.6	34.73	**230.90**	16.5	103.48
		160	152.45	187.15	195.65	205.90	8142.8	46.15	196.15	360.5	37.49	**207.95**	21.5	130.28
15	15	45	64.65	114.10	120.80	125.75	5953.7	13.43	121.40	322.2	8.92	**128.70**	12.6	31.36
		90	73.50	104.90	110.85	**116.85**	6244.0	14.53	111.20	226.2	9.97	116.80	7.9	33.80
	20	60	92.80	154.50	162.60	168.70	7352.8	24.02	162.80	265.7	15.23	**174.30**	13.6	58.04
		120	100.10	139.60	147.70	156.55	8449.1	25.29	148.65	383.9	20.45	**157.20**	13.1	67.50
	40	120	200.40	306.80	322.15	331.30	9009.3	97.21	322.30	403.6	73.18	**349.70**	23.8	244.19
		240	214.75	285.65	298.75	312.50	9341.8	102.04	298.90	280.8	72.52	**320.60**	27.5	266.24
20	20	80	96.10	202.20	214.80	222.90	8944.5	44.54	214.90	296.8	26.81	**231.60**	18.3	107.25
		160	121.50	188.50	199.10	210.10	9012.8	43.49	199.90	345.0	30.49	**211.90**	14.5	102.28
	40	160	219.35	414.05	430.85	441.60	8939.7	168.80	430.40	500.1	129.43	**465.60**	24.1	410.79

Table 6: *experiments on large incomplete grid graphs*

Graph		LB	UB	BFS	PBFS	MaxPBFS	SA			ABC			GA		
n	m			solution	solution	solution	solution	iter.	time	solution	iter.	time	solution	iter.	time
10	10	58	66	20	56	59	59	275	3.10	59	37	1.41	**61**	2	8.56
	15	95	100	30	84	89	91	8583	6.30	89	0	2.97	**93**	5	17.46
	20	124	133	40	112	120	123	9706	10.82	120	643	8.33	**124**	4	23.22
	40	248	266	80	227	242	241	9489	41.13	243	122	22.87	**250**	48	95.94
15	15	145	150	30	125	133	128	584	13.50	133	0	5.88	**144**	5	29.41
	20	195	200	40	170	181	178	9851	23.44	181	1383	25.39	**194**	5	40.96
	40	395	400	80	354	376	356	8486	91.13	376	0	40.16	**395**	8	187.70
20	20	254	266	40	225	237	236	8385	42.12	238	388	26.79	**254**	8	86.47
	40	514	533	80	460	488	473	8994	165.19	488	0	69.54	**514**	7	323.07

Table 7: *experiments on large complete grid graphs*

References

[1] D. Karaboga and B. Basturk, “A powerful and efficient algorithm for numerical function optimization: Artificial Bee Colony (ABC) algorithm”, *Journal of Global Optimization*, vol. 39 , issue 3, pp. 459-471, 2007.

[2] Melanie Mitchell, *An Introduction to Genetic Algorithms (Complex Adaptive Systems)*, The MIT Press, 1998.

[3] Alok Singh, “An artificial bee colony algorithm for the leaf-constrained minimum spanning tree problem”, *Applied Soft Computing*, vol. 9, issue 2, pp. 625–631, 2009.

[4] Tetsuya Fujie, “An exact algorithm for the maximum leaf spanning tree problem”, *Computers & Operations Research*, vol. 30, pp. 1932–1944, 2003.

[5] Lu H and Ravi R, “Approximating maximum leaf spanning trees in almost linear time”, *Journal of Algorithms*, vol. 29, pp. 132-41, 1998.

[6] Solis-Oba, “2-approximation algorithm for finding a spanning tree with maximum number of leaves”, *Lecture notes in computer science*, val. 1461, pp. 441-52, 1998.

[7] M. R. Garey, D. S. Johnson, “Computers and Intractability: A guide to the theory of NP-completeness”, *W. H. Freeman, San Francisco*, 1979.

[8] G. Galbiati, F. Maffioli, A. Morzenti, “A short note on the approximability of the maximum leaves spanning tree problem”, *Information Processing Letters 52*, pp. 45-49, 1994.