ARTICLE IN PRESS

Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Computers & Operations Research ▮▮▮ (▮▮▮▮) ▮▮▮–▮▮▮

computers & operations research

www.elsevier.com/locate/dsw

New metaheuristic approaches for the edge-weighted k-cardinality tree problem

Christian Blum[a,*], Maria J. Blesa[b]

[a]*IRIDIA, Université Libre de Bruxelles, Av. Franklin Roosevelt 50, CP 194/6, B-1050 Brussels, Belgium*
[b]*Dept. LSI, Universitat Politècnica de Catalunya, Jordi Girona 1-3, C6209 Campus Nord, E-08034 Barcelona, Spain*

Abstract

In this paper we propose three metaheuristic approaches, namely a Tabu Search, an Evolutionary Computation and an Ant Colony Optimization approach, for the edge-weighted k-cardinality tree (KCT) problem. This problem is an NP-hard combinatorial optimization problem that generalizes the well-known minimum weight spanning tree problem. Given an edge-weighted graph $G=(V,E)$, it consists of finding a tree in G with exactly $k \leqslant |V|-1$ edges, such that the sum of the weights is minimal. First, we show that our new metaheuristic approaches are competitive by applying them to a set of existing benchmark instances and comparing the results to two different Tabu Search methods from the literature. The results show that these benchmark instances are not challenging enough for our metaheuristics. Therefore, we propose a diverse set of benchmark instances that are characterized by different features such as density and variance in vertex degree. We show that the performance of our metaheuristics depends on the characteristics of the tackled instance, as well as on the cardinality. For example, for low cardinalities the Ant Colony Optimization approach is best, whereas for high cardinalities the Tabu Search approach has advantages.
© 2003 Elsevier Ltd. All rights reserved.

Keywords: k-Cardinality tree problem; Metaheuristics; Combinatorial optimization; Ant Colony Optimization; Evolutionary computation; Tabu search; Spanning trees

1. Introduction

The edge-weighted k-cardinality tree (KCT) problem[1] is a combinatorial optimization problem which generalizes the well-known minimum weight spanning tree problem. It consists of finding in an edge-weighted graph $G=(V,E)$ a subtree with exactly k edges, such that the sum of the weights

[*] Corresponding author. Fax: +32-2-650-2715.
E-mail addresses: cblum@ulb.ac.be (C. Blum), mjblesa@lsi.upc.es (M.J. Blesa).
[1] Also referred to as the *k-minimum spanning tree* (k-MST) problem, or just the *k-tree* problem.

0305-0548/$ - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2003.11.007

is minimal. The problem was first described in [1] and it has gained considerable interest in recent years due to various applications, e.g. in oil-field leasing [2], facility layout [3,4], open pit mining [5], matrix decomposition [6,7], quorum-cast routing [8] and telecommunications [9].

More formally, the KCT problem can be defined as follows. Let $G=(V,E)$ be a graph with a weight function $w: E \rightarrow \mathbb{N}^{+}$ on the edges. We denote by $\mathcal{T}_k$ the set of all k-cardinality trees in G. Then, the edge-weighted problem (G,w,k) consists of finding a k-cardinality tree $T_k \in \mathcal{T}_k$ that minimizes

$$f(T_k) = \sum_{e \in E(T_k)} w(e), \tag{1}$$

where $E(T_k)$ denotes the edges of T_k. Several authors have proved independently that the edge-weighted KCT problem is NP-hard [10,11]. In [11] it has been shown that it is still NP-hard if $\forall e \in E$ it holds that $w(e) \in \{1,2,3\}$, or if $G = K_n$ (the fully connected graph on n vertices). However, the problem is polynomially solvable if there are only two distinct weights. Several authors have considered special types of graphs. One of the results is that the problem is polynomially solvable if G is a tree [12]. The problem is also NP-hard for planar graphs and for points in the plane [11,13]. Polynomial algorithms exist for the cases when all points lie on the boundary of a convex region and for graphs with bounded tree-width [11].

1.1. Existing approaches

In the area of exact methods, a Branch and Cut algorithm based on detailed studies of the associated polyhedron has been developed and implemented in [14]. A Branch and Bound method is also described in [8]. In [15], the number of connected components with k vertices is investigated. Under the assumption that the kth power of the maximum vertex degree of the graph is bounded from above by a polynomial, all the connected components with k vertices can be enumerated in polynomial time. Then, by enumerating the search space, the minimum-cost spanning tree of cardinality k can be found in polynomial time. This approach, however, has never been implemented.

A good range of heuristics has been proposed in [16]. The heuristics mentioned there are based on greedy and dual greedy strategies and also make use of dynamic programming. The implementations of most of the heuristics in [16] are documented in [17]. Other constructive heuristics have been presented in [8].

Lately, authors have begun to apply metaheuristics [18] to the edge-weighted KCT problem. Metaheuristics include, but are not restricted to, Simulated Annealing (SA), Evolutionary Computation (EC), Tabu Search (TS), Ant Colony Optimization (ACO), and Iterated Local Search (ILS), in order of invention. Among the metaheuristic applications to tackle the edge-weighted KCT problem there is an EC approach that uses local search methods to improve solutions [19], two TS methods [20–22], a Variable Neighborhood Search (VNS) approach [23] and an ACO approach [24].

1.2. Our contribution

First, we define a simple neighborhood structure that can be very efficiently computed. This neighborhood structure defines for every solution a neighborhood that is a subset of the neighborhood defined by the neighborhood structure proposed in [21] and used in [20]. Then, we propose three different metaheuristic approaches that make use of this neighborhood structure in different ways. The Tabu Search approach that we propose uses the neighborhood structure for performing moves, whereas the Evolutionary Computation and the Ant Colony Optimization approach incorporate black-box local search procedures that are also based on this neighborhood structure. Furthermore, our Tabu Search approach is characterized by a scheme for dynamically changing the length of the tabu lists. Our EC approach makes use of two heuristically guided crossover operators that were proposed in [25] to solve the node-weighted version of the KCT problem. Finally, our ACO approach is an improved version of the ACO approach that was proposed in [24].

Research on metaheuristics for the edge-weighted KCT problem started in 1995, when the first metaheuristic approach was proposed. Until now, there is no set of commonly accepted and challenging benchmark instances. Furthermore, the approaches that were proposed so far were only tested on d-regular graphs (i.e., in a d-regular graph each vertex is connected to d other vertices), even though it was discovered before that the KCT problem appears to be especially hard to solve for grid graphs. However, in order to be able to compare our algorithms with already existing approaches, we first applied them to a set of 4-regular benchmark instances provided by Blesa and Xhafa in [20,26], who generated these instances using a software tool by Jörnsten and Løkketangen [21]. Then, we developed a diverse set of problem instances with the following distinguishing features: (i) size of the graph (number of nodes, respectively edges), (ii) sparsity, respectively density, of the graph, and (iii) the variance of the vertex degrees (i.e., a high variance of the vertex degrees may be an indicator for the clusteredness of the graph). We conducted a considerable amount of experiments on a homogeneous cluster of computers in order to compare the results obtained by our three metaheuristic approaches.

The remainder of the paper is organized as follows. In Section 2 we describe the neighborhood structure which is used by all metaheuristic approaches in different ways. Sections 3–5 contain the detailed descriptions of our metaheuristic approaches to tackle the KCT problem. In Section 6 we propose a new set of diverse benchmark instances and we perform an experimental evaluation of our methods. Finally, Section 7 offers conclusions and an outlook to the future.

2. Common neighborhood structure

Let $\mathscr{S}$ denote the search space. Then, a neighborhood structure $\mathscr{N}: \mathscr{S} \mapsto 2^{\mathscr{S}}$ is a function that provides a set of neighbors for every solution $s \in \mathscr{S}$. The neighborhood structure we chose for the KCT problem is very simple and intuitive. The neighborhood $\mathscr{N}_{\text{leaf}}(T_k)$ of a k-cardinality tree T_k consists of all k-cardinality trees which can be generated by removing a leaf edge e, which results in a $(k-1)$-cardinality tree T_{k-1}, and adding an edge from the set $E_{\text{NH}}(T_{k-1}) \setminus \{e\}$, where $E_{\text{NH}}(T_{k-1})$ is defined as follows:

$$E_{\text{NH}}(T_{k-1}) \leftarrow \{e = \{v, v'\} \in E(G) \mid v \in V(T_{k-1}) \ \textbf{XOR} \ v' \in V(T_{k-1})\}, \tag{2}$$

where $E(G)$ denotes the edges of graph G. Intuitively, set $E_{\text{NH}}(T_{k-1})$ consists of all edges that do not belong to T_{k-1} and that have exactly one end-point in T_{k-1}. We are going to use this neighborhood structure in the TS approach for performing moves and in the population-based methods in black-box local search procedures for improving solutions. This simple neighborhood can be very efficiently generated since both, the set of leaves of a k-cardinality tree as well as set $E_{\mathrm{NH}}(T_{k-1})$, can be updated incrementally during the search process without the necessity to recompute them from scratch. Note that the neighborhood structure that was proposed in [21] and also used in [20] for Tabu Search approaches additionally considers so-called cyclic moves. A cyclic move is generated by adding to the current k-cardinality tree an edge that produces a cycle and by removing a different edge from this cycle such that the result is again a tree. However, the computational expenses of this neighborhood are much higher.

3. Tabu Search approach

Tabu Search (TS) is among the most cited and used metaheuristics for the application to combinatorial optimization problems [27,28]. TS is based on local search. Basic local search is usually called *iterative improvement*, since each move[2] within a given neighborhood structure $\mathcal{N}(\cdot)$ is only performed if the solution it produces is better than the current solution. The iterative improvement procedure stops as soon as it finds a local minimum. The improvement performed can either be a *first improvement*, or a *best improvement*. The former scans the neighborhood $\mathcal{N}(s)$ of a solution s and chooses the first solution with a lower objective function value than s, the latter exhaustively explores the neighborhood and returns one of the solutions with the lowest objective function value.

TS explicitly uses the history of the search, both to escape from local minima and to implement diversification and intensification strategies. The basic algorithm applies a best improvement local search as basic ingredient and uses a *short-term memory* to escape from local minima and to avoid cycling. The short-term memory is implemented as a set of *tabu lists* that store solution *attributes*. Attributes refer usually to components of solutions, moves, or differences between two solutions. The contents of the tabu lists defines the *tabu conditions* which are used to filter the neighborhood of a solution and generate the *allowed set*, which is a subset of the set of neighbors. The use of tabu lists prevents the algorithm from returning to recently visited solutions. Therefore, it may enforce to accept even non-improving moves. The length of the tabu lists (i.e., the *tabu list tenure*) determines the behavior of the algorithm. With small tabu tenures the search will concentrate on limited areas of the search space. On the opposite, a larger tabu tenure forces the search process to explore larger regions, because it forbids revisiting a higher number of solutions. The tabu tenure can be varied during the search process.

Our TS approach to tackle the edge-weighted KCT problem uses two tabu lists, henceforth denoted by *InList* and *OutList*. The attributes they store are the edges that were recently added, respectively removed, in the search process. Every move involves removing one edge $e \in T_k^{\mathrm{cur}}$ from the current k-cardinality tree T_k^{cur}, and adding a different edge to $T_k^{\mathrm{cur}} - e$. *InList* is the list to keep memory of the removed edges, respectively *OutList* is the list to store added edges.

Another characterizing feature of our approach is the use of a dynamic tabu list tenure tl_{ten}. Depending on the problem instance to be tackled, a minimum tabu list tenure tt_{min},

[2] A move is the transition from a solution s to a solution $s' \in \mathcal{N}(s)$ and usually defined by the modification which has to be done to s in order to generate s'.

a maximum tabu list tenure $tt_{\max}$ and an increment value tt_{inc} are computed. At the beginning of every restart phase (i.e., the current solution is deleted and a new initial solution is generated), tl_{ten} is set to $tt_{\min}$. If the restart-best solution T_k^{rb} was not improved for a maximum number of $nic_{\max}$ iterations, the tabu list tenure tl_{ten} is increased by tt_{inc} in order to diversify the search process. Whenever the restart-best solution T_k^{rb} is improved, the tabu list tenure tl_{ten} is set back to $tt_{\min}$ in order to intensify the search process around T_k^{rb}. In case the increase of the tabu list tenure would result in a tabu list tenure greater than $tt_{\max}$, a restart is performed. This can be regarded as an escaping mechanism for situations when the search process seems to be stuck.

The framework of our TS approach to tackle the edge-weighted KCT problem is shown in Algorithm 1. We will refer to this algorithm as TS_KCT. In the following, the components of this algorithm are explained in more detail.

```
Algorithm 1 TS for the KCT problem (TS_KCT)
input: a problem instance $(G, w, k)$
InitializeParameters($tt_{\min}, tt_{\max}, tt_{\text{inc}}, tl_{\text{ten}}, nic, nic_{\max}$)
InitializeTabuLists($InList, OutList, tl_{\text{ten}}$)
$T_k^{\text{cur}} \leftarrow$ GenerateInitialSolution()
$T_k^{\text{gb}} \leftarrow T_k^{\text{cur}}$, $T_k^{\text{rb}} \leftarrow T_k^{\text{cur}}$
while termination conditions not met do
    $T_k^{\text{new}} \leftarrow$ FirstImprovingNeighbor($\mathcal{N}_{\text{leaf}}(T_k^{\text{cur}}), InList, OutList$)
    if $T_k^{\text{new}} \neq$ NULL then
        UpdateTabuLists($T_k^{\text{cur}}, T_k^{\text{new}}, InList, OutList$)
        $T_k^{\text{cur}} \leftarrow T_k^{\text{new}}$
        Update($T_k^{\text{cur}}, T_k^{\text{rb}}, T_k^{\text{gb}}, nic$)
        if $nic > nic_{\max}$ then
            if $tl_{\text{ten}} + tt_{\text{inc}} > tt_{\max}$ then
                PerformRestart()
            else
                $tl_{\text{ten}} \leftarrow tl_{\text{ten}} + tt_{\text{inc}}$
            end if
        end if
    else
        PerformRestart()
    end if
end while
output: $T_k^{\text{gb}}$
```

InitializeParameters($tt_{\min}, tt_{\max}, tt_{\text{inc}}, tl_{\text{ten}}, nic, nic_{\max}$): After preliminary tests, we decided to set the algorithm parameters depending on the problem instance $(G = (V, E), w, k)$ under consideration in the following way:

$$tt_{\min} \leftarrow \min\left\{ \left\lfloor \frac{|V|}{5} \right\rfloor, |V| - \{k, k\} \right\}$$

$$tt_{\max} \leftarrow \left\lfloor \frac{|V|}{3} \right\rfloor$$

$$tt_{\text{inc}} \leftarrow \left\lfloor \frac{tt_{\max} - tt_{\min}}{4} \right\rfloor + 1$$

$$nic_{\max} \leftarrow \max\{tt_{\text{inc}}, 200\}.$$

The setting of $tt_{\min}$ considers the fact that for small and big cardinalities the tabu list tenure can be rather small, whereas in the middle of the cardinality range a greater tabu list tenure might be required since the complexity of the problem is higher. The setting of $tt_{\max}$ and tt_{inc} is such that the tabu list tenure can be successively increased for four times before a restart is performed. The minimum value of $nic_{\max}$, the maximum number of iterations without improvement, is set to 200, and *nic* the counter for the number of iterations without improvement is initialized to 0.

InitializeTabuLists($InList, OutList, tl_{\text{ten}}$): In this procedure, the two tabu lists are initialized to empty lists, and they are given a maximum size of tl_{ten}.

GenerateInitialSolution(): To construct a k-cardinality tree T_k, first, an edge $e = \{v, v'\} \in E$ is chosen uniformly at random. With this edge, a 1-cardinality tree T_1 with edge e and two nodes v and v' is created. Then, a k-cardinality tree T_k is constructed in a greedy manner by adding at each of $k-1$ construction steps an edge $e \leftarrow \text{argmin}\{w(e') \mid e' \in E_{\text{NH}}(T_t)\}$ to the current t-cardinality tree T_t under construction, where $t \in \{1, \ldots, k-1\}$.

FirstImprovingNeighbor($\mathcal{N}_{\text{leaf}}(T_k^{\text{cur}}), InList, OutList$): In this function, a neighbor of the current k-cardinality tree T_k^{cur} is chosen. Only neighbors from the allowed set are eligible to be chosen. The allowed set of neighbors is defined as follows. A k-cardinality tree $T_k \in \mathcal{N}_{\text{leaf}}(T_k^{\text{cur}})$, where $T_k = T_k^{\text{cur}} - e_{\text{out}} + e_{\text{in}}$, is eligible, if and only if

(1) $e_{\text{in}} \notin InList$ and $e_{\text{out}} \notin OutList$ (otherwise T_k is called *tabu*), or
(2) $f(T_k) < f(T_k^{\text{gb}})$, where T_k^{gb} is the best solution found since the start of the algorithm;

The second condition is called an *aspiration criterion*. Aspiration criteria are necessary because storing only attributes in the tabu lists introduces a loss of information, since forbidding a move means assigning the tabu-status to probably more than one solution. Thus, it is possible that unvisited solutions of good quality are excluded from the allowed set.

Since our TS approach is a *first improvement* approach, the neighborhood exploration (i.e., the exploration of the allowed set) stops when the first eligible neighbor that has a lower objective function value than the current solution is encountered. Otherwise the whole neighborhood is scanned and the best neighbor solution, which has in this case a higher objective function value than the current solution, is chosen. If no eligible neighbor can be found, then this function returns NULL. The order in which neighbors are examined is the following one: The leaves of the current tree T_k^{cur} are examined in order of decreasing weight. In contrast, the candidates to join the current tree T_k^{cur} are examined in order of increasing weight. This way of exploring the neighborhood has the advantage that neighbors with a lower objective function value are likely to be found earlier in the exploration process. For example, if it is possible to remove the highest weighted leaf and to add the lowest weighted candidate to join the current tree, one can be sure that there is no better neighbor available.

UpdateTabuLists($T_k^{\mathrm{cur}}, T_k^{\mathrm{new}}, \mathit{InList}, \mathit{OutList}$): After a neighbor $T_k^{\mathrm{new}} = T_k^{\mathrm{cur}} - e_{\mathrm{out}} + e_{\mathrm{in}}$ has been chosen, the tabu lists *InList* and *OutList* are updated, i.e., e_{in} is added to *OutList* and e_{out} is added to *InList*. Both tabu lists work as first-in-first-out lists with a given length tl_{ten}. The tabu lists have the effect that an edge that just recently was removed from (respectively, has entered) the current tree, cannot enter (respectively, be removed from) it in the near future.

Update($T_k^{\mathrm{cur}}, T_k^{\mathrm{rb}}, T_k^{\mathrm{gb}}, nic$): If $f(T_k^{\mathrm{cur}}) < f(T_k^{\mathrm{rb}})$, then T_k^{rb} is set to T_k^{cur}. The same is done for T_k^{gb}. In contrast, if $f(T_k^{\mathrm{cur}}) \geqslant f(T_k^{\mathrm{rb}})$ then *nic* is increased by one, where *nic* is the counter of the number of successive iterations in which the restart-best solution was not improved.

PerformRestart(): If there is no eligible neighbor, or if the tabu list tenure tl_{len} has exceeded its maximum $tt_{\max}$, a restart is performed as shown in Algorithm 2.

```
Algorithm 2 PerformRestart()
$tl_{\mathrm{ten}} \leftarrow tt_{\min}$
InitializeTabuLists($\mathit{InList}, \mathit{OutList}, tl_{\mathrm{ten}}$)
$T_k^{\mathrm{cur}} \leftarrow$ GenerateInitialSolution()
$T_k^{\mathrm{rb}} \leftarrow T_k^{\mathrm{cur}}$
$nic \leftarrow 0$
```

4. Evolutionary Computation approach

Evolutionary Computation (EC) algorithms [29–31] are widely used to tackle *NP*-hard combinatorial optimization problems. They are inspired by nature's capability to evolve living beings which are well adapted to their environment. EC algorithms can shortly be characterized as computational models of evolutionary processes working on populations of individuals. Individuals are in most cases solutions to the problem under consideration. Usually EC algorithms apply operators, which are called *recombination* or *crossover* operators, to recombine two or more individuals to produce new individuals. In addition to that, operators that cause a self-adaptation of individuals are applied. These operators are called *mutation* or *modification* operators (depending on their structure). The driving force in evolutionary algorithms is the *selection* of individuals based on their *fitness*. Individuals with a higher fitness have a higher probability to be chosen as members of the next iterations' population (or as parents for producing new individuals). This principle is called *survival of the fittest* in natural evolution. It is the capability of nature to adapt itself to a changing environment which gave the inspiration for EC algorithms.

Our EC approach to tackle the edge-weighted KCT problem is characterized by (i) the use of two heuristically guided crossover operators, by (ii) an ageing mechanism for individuals and by (iii) the use of black-box local search procedures based on the neighborhood $\mathcal{N}_{\mathrm{leaf}}$ as described in Section 2 to improve individuals. The framework of our algorithm is shown in Algorithm 3, where P denotes a population of n individuals, T_k^{gb} denotes the best k-cardinality tree found so far in the course of the search, and T_k^{ib} denotes the iteration-best solution. We will refer to this algorithm as EC_KCT. Its components are outlined in the following.

```
Algorithm 3 EC for the KCT problem (EC_KCT)
input: a problem instance $(G, w, k)$
$T_k^{gb} \leftarrow NULL$
$n \leftarrow$ DeterminePopulationSize$(G, k)$
$P \leftarrow$ GenerateInitialPopulation$(n)$
while termination conditions not met do
    $P \leftarrow$ ApplyCrossover$(P)$
    $P \leftarrow$ ApplyLocalSearch$(P)$
    $P \leftarrow$ ApplyEliteAction$(P)$
    $T_k^{ib} \leftarrow \mathrm{argmin}\{f(T_k) \mid T_k \in P\}$
    Update$(T_k^{gb}, T_k^{ib})$
    IncreaseAgeOfIndividuals$(P)$
    $P \leftarrow$ RemoveOverAgedIndividuals$(P)$
    $P \leftarrow P \cup$ GenerateRandomIndividuals$(n - |P|)$
end while
output: $T_k^{gb}$
```

DeterminePopulationSize(G, k): The population size n is set to $\lfloor |E|/k \rfloor$ and is therefore a function of $|E|$ and k. In general, for smaller cardinalities bigger populations are needed, because the probability that trees overlap (i.e., have edges in common) is smaller. This is important, as only k-cardinality trees that overlap can be crossover partners (see below). We set the minimum population size to 50 and the maximum population size to 200.

GenerateInitialPopulation(n): The algorithm is initialized with a population of randomly generated k-cardinality trees. This is in contrast to the initial solution in TS_KCT, which was created in a greedy manner. The reason is that we experimentally found a random initial population to give better results than an initial population that was created in a greedy manner. Furthermore, all individuals have an age that is initialized to 0 at the time of creation, and the age of an individual is incremented in case it enters the next generation (later we are going to outline events that reset the age of individuals to 0). The age of individuals is used to determine if an individual is still useful or not, i.e., when the age limit is reached we assume the individual not to be useful anymore.

ApplyCrossover(P): For every k-cardinality tree T_k in the current population, a partner T_k^{p} for crossover is chosen among those individuals in P which have at least one edge in common with T_k. The choice is done in a roulette-wheel-selection manner with respect to the inverse of the objective function value $f(\cdot)$ of a tree. This means that trees of lower weight have a better chance to be chosen as a crossover partner. If there is no individual with at least one edge in common with T_k then no crossover can be performed and T_k does not enter the population of the next generation. Two different heuristically guided crossover operators are applied to the two crossover partners T_k and T_k^{p}. The *U-crossover* (for "Union"-crossover) aims at picking the good properties that distinguish one parent from the other and combining them in the offspring, while the *I-crossover* (for "Intersection"-crossover) aims at preserving the good properties both parents have in common in the offspring and to take the best from the rest. Both operators are, in some sense, complementary to each other.

```
**Algorithm 4** Framework for *U-crossover* and *I-crossover*
**input:** Two $k$-cardinality trees $T_k$ and $T_k^{\mathrm{p}}$
$E_{\cup} \leftarrow E(T_k) \cup E(T_k^{\mathrm{p}})$
$E_{\cap} \leftarrow E(T_k) \cap E(T_k^{\mathrm{p}})$
$t \leftarrow 1$
$E(T_t^{\mathrm{child}}) \leftarrow \{\mathsf{argmin}\{w(e = \{v, v'\}) \mid e \in E_{\cap}\}\}$
$V(T_t^{\mathrm{child}}) \leftarrow \{v, v'\}$
**repeat**
    Choose set $S$ according to equation (3) for $U$-crossover, resp. (4) for $I$-crossover
    **if** $S = \emptyset$ **then**
        $e = \{v, v'\} \leftarrow \mathsf{argmin}\{w(e) \mid e \in E_{\mathrm{NH}}(T_t^{\mathrm{child}}) \cap E_{\cup}\}$
    **else**
        $e = \{v, v'\} \leftarrow \mathsf{argmin}\{w(e) \mid e \in S\}$
    **end if**
    $E(T_t^{\mathrm{child}}) \leftarrow E(T_t^{\mathrm{child}}) \cup \{e\}$
    $V(T_t^{\mathrm{child}}) \leftarrow V(T_t^{\mathrm{child}}) \cup \{v, v'\}$
    $t \leftarrow t + 1$
**until** $|E(T_t^{\mathrm{child}})| = k$
**output:** $T_k^{\mathrm{child}}$
```

The algorithmic framework for both operators is given in Algorithm 4, in which T_t^{child} denotes the partial k-cardinality tree at construction step t, which is constructed as the child of the input k-cardinality trees T_k and T_k^{p}. It remains to be specified how set S (see Algorithm 4) is generated in each case of crossover. In the case of *U-crossover*, S in construction step t is defined as follows:

$$S \leftarrow E_{\mathrm{NH}}(T_t^{\mathrm{child}}) \cap (E_{\cup} \setminus E_{\cap}), \tag{3}$$

where $E_{\mathrm{NH}}(T_t^{\mathrm{child}})$ consists of all edges in G with exactly one end-point in T_t^{child} (see Eq. (2)), and $E_{\cup}$ and $E_{\cap}$ are the union and intersection of edges as defined in Algorithm 4. In *U-crossover*, edges with a low weight that appear only in one of the two parents are preferred. On the contrary, in *I-crossover* low-weighted edges are preferred that are common to both parents by defining S in construction step t as follows:

$$S \leftarrow E_{\mathrm{NH}}(T_t^{\mathrm{child}}) \cap E_{\cap}, \tag{4}$$

where $E_{\mathrm{NH}}(T_t^{\mathrm{child}})$ is as described above. After producing two offspring with parents T_k and T_k^{p}, the best tree among the first parent T_k and the two offspring is chosen to enter the next population. This means that the first parent tree is only replaced if at least one of the two offspring is better. Furthermore, only those individuals which are pairwise different to all the other individuals already in the population are introduced.

ApplyLocalSearch(P): To every individual produced by the crossover procedure we apply the *best improvement* local search based on the neighborhood structure $\mathcal{N}_{\mathrm{leaf}}$ as outlined in Section 2. If the application of this local search improves an individual, we regard it as new by setting its age back to 0.

ApplyEliteAction(P): As an elite action we apply a short run of our TS approach introduced in Section 3 to the best individual of the current population. The duration of this TS run was chosen to be $2 \cdot k$ iterations.

Update($T_k^{\text{gb}}, T_k^{\text{ib}}$): In this function we update T_k^{gb}, which is the best solution found since the start of the algorithm, with the iteration-best solution T_k^{ib}, i.e., T_k^{ib} replaces T_k^{gb} if $f(T_k^{\text{ib}}) < f(T_k^{\text{gb}})$.

IncreaseAgeOfIndividuals(P): In this procedure the age of each individual of the new population P is incremented.

RemoveOverAgedIndividuals(P): This procedure is applied to remove individuals that exceed the age limit. For all the experiments we chose 10 as the age limit.

GenerateRandomIndividuals($n - |P|$): There are several events that cause a population to shrink. The first one is that no crossover partner can be found for an individual. In this case this individual does not enter the population of the next generation. Furthermore, we take care that the individuals of a population are pairwise different, which means in practise that an individual produced by crossover or the application of local search does not enter the next population if an equivalent individual already exists in the population under construction. In order to keep the population size fixed, we apply procedure GenerateRandomIndividuals($n-|P|$) to generate $n-|P|$ random individuals as described in the generation of the initial population. These random individuals are then inserted into the population.

In summary, our EC approach can be regarded as a population of interacting hill-climbers that are replaced by new randomly generated hill-climbers once they reach their age limit.

5. Ant Colony Optimization approach

Ant Colony Optimization (ACO) [32–34] is a metaheuristic approach for solving hard combinatorial optimization problems. The inspiring source of ACO is the foraging behavior of real ants. While walking from food sources to the nest and vice versa, ants deposit a substance called *pheromone* on the ground. Paths marked by strong pheromone concentrations are more probable to be chosen when deciding about a direction to go. This basic behavior is the basis for a cooperative interaction which leads to the emergence of shortest paths, thus minimizing the length of the path between nest and food source.

In ACO algorithms, an artificial ant incrementally constructs a solution by adding opportunely defined solution components to a partial solution under construction.[3] The solution components to be added are chosen probabilistically with respect to a parametrized probabilistic model, the so-called *pheromone model*. A parametrized probabilistic model is specified by a set $\mathcal{T}$ of model parameters. In ACO algorithms, these model parameters are called *pheromone trail* parameters. The values of the pheromone trail parameters are called *pheromone values*. The first ACO algorithm proposed was Ant System (AS) [34]. In recent years some changes and extensions of AS have been proposed, e.g., Ant Colony System (ACS) [35] and $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) [36].

Our approach is a $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System implemented in the Hyper-Cube Framework [37]. Implementing ACO algorithms in the Hyper-Cube Framework comes with several benefits.

[3] Therefore, the ACO metaheuristic can be applied to any combinatorial optimization problem for which a constructive heuristic can be defined.

An important one is the automatic scaling of the objective function. The basic framework of our algorithm is shown in Algorithm 5, in which $\mathscr{T}$ is a set of pheromone trail parameters, n_a is the number of ants used in every iteration, T_k^j are solutions to the problem, and *cf* is a numerical value that is called the convergence factor. Furthermore, T_k^{ib} denotes the iteration-best solution, T_k^{rb} is the restart-best solution, and T_k^{gb} is the best solution found from the start of the algorithm. We will refer to this algorithm as ACO_KCT. Its components are outlined in the following.

```
Algorithm 5 ACO for the KCT problem (ACO_KCT)
input: a problem instance $(G, w, k)$
$T_k^{gb} \leftarrow NULL,\ T_k^{rb} \leftarrow NULL,\ cf \leftarrow 0,\ global\_conv \leftarrow FALSE$
$n_a \leftarrow$ DetermineNumberOfAnts$(G, k)$
InitializePheromoneValues($\mathscr{T}$)
while termination conditions not met do
    for $j = 1$ to $n_a$ do
        $T_k^j \leftarrow$ ConstructSolution($\mathscr{T}$)
        LocalSearch($T_k^j$)
    end for
    $T_k^{ib} \leftarrow$ argmin$\{f(T_k^1), \ldots, f(T_k^{n_a}\}$
    $T_k^{ib} \leftarrow$ ApplyEliteAction($T_k^{ib}$)
    ApplyPheromoneValueUpdate($cf, global\_conv, \mathscr{T}, T_k^{ib}, T_k^{rb}, T_k^{gb}$)
    Update($T_k^{ib}, T_k^{gb}, T_k^{rb}$)
    $cf \leftarrow$ ComputeConvergenceFactor($\mathscr{T}, T_k^{ib}$)
    if $cf \geqslant 0.99$ AND $global\_conv = TRUE$ then
        ResetPheromoneValues($\mathscr{T}$)
        $T_k^{rb} \leftarrow NULL$
        $global\_conv \leftarrow FALSE$
    else
        if $cf \geqslant 0.99$ then $global\_conv \leftarrow TRUE$
    end if
end while
output: $T_k^{gb}$
```

DetermineNumberOfAnts(G, k): The number of ants n_a is (as the population size in the EC approach) set to $\lfloor |E|/k \rfloor$ and is therefore a function of $|E|$ and k. We set the minimum number of ants to 15 and the maximum number of ants to 50. These limits are lower than the limits for the population size in the EC approach because the philosophy of our ACO approach is to focus quickly on a certain area in the search space and to get the global perspective by performing restarts.

InitializePheromoneValues($\mathscr{T}$): We initialize the pheromone values to 0.5 because in the Hyper-Cube Framework the pheromone values are limited to $[0, 1]$, and therefore this setting gives equal chance to both directions.

ConstructSolution($\mathscr{T}$): To build a solution, an ant starts from a randomly chosen edge and does $k-1$ construction steps as shown in Algorithm 6. At each step of the construction phase an edge $e \in E_{\text{NH}}(T_{t-1})$ (see Eq. (2)) is added to the $(t-1)$-cardinality tree T_{t-1}, where $t \in \{2, \ldots, k\}$. The choice of the next edge to be added depends on the underlying pheromone model. The model that we

```
Algorithm 6 Ant construction phase
Choose an edge $e^* = \{v, v'\} \in E$ with probability $\mathbf{p}(e^*) = \frac{\tau_{e^*}}{\sum_{e \in E} \tau_e}$
$E(T_1) \leftarrow \{e^*\}$
$V(T_1) \leftarrow \{v, v'\}$
for $t = 2$ to $k$ do
    $e^* = \{v, v'\} \leftarrow \mathsf{ChooseNextEdge}(E_{\mathrm{NH}}(T_{t-1}))$
    $E(T_t) \leftarrow E(T_{t-1}) \cup \{e^*\}$
    $V(T_t) \leftarrow V(T_{t-1}) \cup \{v, v'\}$
end for
```

chose consists of a pheromone trail parameter $\mathcal{T}_e$ with pheromone value τ_e for every edge $e \in E(G)$. Therefore, if $|E(G)| = m$ we have m pheromone trail parameters. Given the set $E_{\mathrm{NH}}(T_{t-1})$, the next edge is chosen in function $\mathsf{ChooseNextEdge}(E_{\mathrm{NH}}(T_{t-1}))$ as follows: we draw a random number p between 0 and 1 and, if $p \leqslant 0.8$, the next edge e^* is chosen deterministically:

$$e^* \leftarrow \mathsf{argmin} \left\{ \tau_e \frac{1}{w(e)} \,\middle|\, e \in E_{\mathrm{NH}}(T_{t-1}) \right\}. \tag{5}$$

Otherwise the next edge e^* is chosen probabilistically with the following transition probabilities:

$$\mathbf{p}(e \mid T_{t-1}) = \begin{cases} \dfrac{\tau_e / w(e)}{\sum_{e' \in E_{\mathrm{NH}}(T_{t-1})} \tau_{e'} / w(e')} & \text{if } e \in E_{\mathrm{NH}}(T_{t-1}), \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

Eq. (6) shows that the transition probabilities are biased by the weights of the edges: the lower the weight of an edge is, the higher its probability to be chosen. With this pheromone model the algorithm tries to learn for every edge the desirability of having it in a solution. The fact of making in 80% of the construction steps a deterministic decision leads the algorithm already at the beginning of the search to relatively good areas in the search space. However, there is the danger that the algorithm gets stuck more easily in local optima. On the other side, the advantage is a more efficient utilization of computation time, which is needed in order for the algorithm to be competitive with the EC and the TS approach on big problem instances.

$\mathsf{LocalSearch}(T_k^j)$: A *best improvement* local search based on the neighborhood structure $\mathcal{N}_{\mathrm{leaf}}$ as outlined in Section 2 is applied to every k-cardinality tree T_k^j produced by the ants.

$\mathsf{ApplyEliteAction}(T_k^{\mathrm{ib}})$: A short run of the TS approach as outlined in Section 3 is applied to T_k^{ib} (the best k-cardinality tree of each iteration) in order to further improve this solution. The duration of the run was chosen to be $2 \cdot k$ iterations, as for the elite action in the EC approach.

$\mathsf{ApplyPheromoneUpdate}(cf, global\_conv, \mathcal{T}, T_k^{\mathrm{ib}}, T_k^{\mathrm{rb}}, T_k^{\mathrm{gb}})$: Three different solutions are used for updating the pheromone values: (i) the best solution found in the current iteration T_k^{ib}, (ii) the restart-best solution T_k^{rb} and, (iii) the best solution found since the start of the algorithm T_k^{gb}. A particularity is that the influence of each one of these three solutions is made dependent on the state of convergence of the algorithm (given by the convergence factor *cf*) rather than on its objective function value. To perform the update, first an update value ξ_e for every edge $e \in E(G)$ is computed:

$$\xi_e \leftarrow \kappa_{\mathrm{ib}} \delta(T_k^{\mathrm{ib}}, e) + \kappa_{\mathrm{rb}} \delta(T_k^{\mathrm{rb}}, e) + \kappa_{\mathrm{gb}} \delta(T_k^{\mathrm{gb}}, e), \tag{7}$$

Table 1
The schedule used for values ρ, κ_{ib}, κ_{rb} and κ_{gb} depending on *cf* and on the Boolean variable *global_conv*

	$global\_conv = FALSE$			$global\_conv = TRUE$
	$cf < 0.7$	$cf \in [0.7, 0.95)$	$cf \geqslant 0.95$	
ρ	0.15	0.1	0.05	0.1
κ_{ib}	2/3	1/3	0	0
κ_{rb}	1/3	2/3	1	0
κ_{gb}	0	0	0	1

where κ_{ib} is the weight of T_k^{ib}, κ_{rb} the weight of T_k^{rb}, and κ_{gb} the weight of T_k^{gb} such that $\kappa_{\text{ib}} + \kappa_{\text{rb}} + \kappa_{\text{gb}} = 1.0$. The δ-function is defined as follows:

$$\delta(T_k, e) = \begin{cases} 1 & e \in E(T_k), \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

Depending on the convergence factor *cf*, the weight of each of the three solutions is determined. The convergence factor *cf* is a value providing an estimate about the state of convergence of the system. At each iteration, the convergence factor is computed in the following way:

$$cf \leftarrow \frac{\sum_{e \in E(T_k^{\text{ib}})} \tau_e}{k \cdot \tau_{\max}}, \tag{9}$$

where $\tau_{\max}$ is an upper limit for the pheromone values (see below). The convergence factor *cf* can therefore only assume values between 0 and 1. It is clear that if *cf* is close to 1 then the system is in a state where the probability to produce solution T_k^{ib} is close to 1 and therefore the probability to produce a solution different to T_k^{ib} is close to 0. This is what we informally call the *state of convergence* for our system. After preliminary experiments, we chose the schedule of settings for values ρ, κ_{ib}, κ_{rb} and κ_{gb} as shown in Table 1. To all pheromone values τ_e we then apply the following update rule:

$$\tau_e \leftarrow f_{\text{mmas}}(\tau_e + \rho \cdot (\xi_e - \tau_e)), \tag{10}$$

where $\rho \in (0, 1]$ is a constant called learning rate, and

$$f_{\text{mmas}}(x) = \begin{cases} \tau_{\min}, & x < \tau_{\min}, \\ x, & \tau_{\min} \leqslant x \leqslant \tau_{\max}, \\ \tau_{\max}, & x > \tau_{\max}. \end{cases} \tag{11}$$

Remember that by using the Hyper-Cube Framework (see [37]) the pheromone values are limited to $[0, 1]$. Additionally, we introduce a lower bound $\tau_{\min}$ for the pheromone values and set it to 0.001, and an upper bound $\tau_{\max}$, which is set to 0.999. These lower and upper bounds are used to prevent the algorithm from converging to a solution. Therefore, after applying the pheromone update we set those pheromone values that exceed the upper bound back to the upper bound and those that fall below the lower bound back to the lower bound.

At the beginning of the search process the learning rate is set to the value 0.15 (see Table 1), because we want our algorithm to focus quite quickly on an area in the search space. Once the convergence factor *cf* is bigger than 0.7 the learning rate is decreased in order to perform a more careful search, and also the influence of the restart-best solution is increased. Once the algorithm is near the state of convergence, only the restart-best solution is used to update the pheromone values and we decrease the learning rate even more in the hope to find a better solution near the restart-best solution. When the limit for the convergence factor is reached (i.e., $cf \geqslant 0.99$), the best solution found since the start of the algorithm is used to update the pheromone values. The reason behind that is the hope to find a better solution in-between two good solutions which are the restart-best and the overall best solution in this case. Once the limit for the convergence factor is reached again, a restart is performed.

Update($T_k^{\text{ib}}, T_k^{\text{rb}}, T_k^{\text{gb}}$): In this procedure we replace T_k^{rb} with T_k^{ib} if $f(T_k^{\text{ib}}) < f(T_k^{\text{rb}})$. The same is done for T_k^{gb}.

ComputeConvergenceFactor($\mathscr{T}, T_k^{\text{ib}}$): The convergence factor *cf* is re-computed in every iteration according to Eq. (9).

ResetPheromoneValues($\mathscr{T}$): In this procedure all pheromone values τ_e are set back to the starting value 0.5.

6. Experimental comparison

We decided for C++ as the programming language and we compiled all our software with the GNU C++ compiler *gcc*, version 2.95.3. Furthermore, we implemented the three metaheuristic approaches on the same data structures. Finally, all the metaheuristic approaches were tested on a beowulf cluster consisting of 8 identical PCs with AMD Athlon 1100 MHz CPU under RedHat Linux 7.0. As mentioned already in the respective sections, our ACO approach is denoted by ACO_KCT, our EC approach by EC_KCT, and our TS approach by TS_KCT.

First, we show the competitiveness of our algorithms by applying them to existing benchmark instances for which results already exist. In 2001, Blesa and Xhafa used a software tool developed by Jörnsten and Løkketangen [21] for producing 35 edge-weighted 4-regular graphs of different sizes (i.e., the smallest ones on 25 nodes, and the biggest ones on 1000 nodes). Then, for cardinality $k=20$, they applied the Tabu Search algorithm by Jörnsten and Løkketangen [21] and their own Tabu Search algorithm [20] to all these benchmark instances and published the results at [26]. In order to compare with them, we also applied our three algorithms for cardinality $k = 20$ to each of these benchmark instances. The results are shown in Table 2. They show that for 12 out of 35 benchmark instances we improve the best known solution. In the remaining 23 cases our algorithms find the same solution quality as was found either by the Tabu Search by Jörnsten and Løkketangen, or by the Tabu Search by Blesa and Xhafa. But more importantly, the results show that our algorithms find for each problem instance the same best solution in a very short amount of computation time. This means that most of these benchmark instances are too small for showing differences between our algorithms. Furthermore, all the 35 graphs are 4-regular and therefore very special types of graphs. Therefore, we decided to test our algorithms on graphs with different characteristics, and also on a whole range of different cardinalities rather than just on one particular cardinality (i.e., $k = 20$).

Table 2
Comparison of the results obtained by ACO_KCT, EC_KCT, and TS_KCT with the best known solutions that were obtained by either the Tabu Search by Jörnsten and Løkketangen [21], or by the Tabu Search by Blesa and Xhafa [20] on the 35 4-regular benchmark instances (cardinality $k = 20$) taken from [26]

Instance	Best known	ACO_KCT				EC_KCT				TS_KCT				Time limit (s)
		Best	Average	$\sqrt{\sigma^2}$	$\bar{t}$	Best	Average	$\sqrt{\sigma^2}$	$\bar{t}$	Best	Average	$\sqrt{\sigma^2}$	$\bar{t}$	
g25-4-01	219	***219**	219	0	0.004	***219**	219	0	0.084	***219**	219	0	0.004	2
g25-4-02	607	***607**	607	0	0.006	***607**	607	0	0.076	***607**	607	0	0.016	2
g25-4-03	464	***464**	464	0	0.006	***464**	464	0	0.086	***464**	464	0	0.008	2
g25-4-04	620	***620**	620	0	0.01	***620**	620	0	0.073	***620**	620	0	0.024	2
g25-4-05	573	***573**	573	0	0.017	***573**	573	0	0.087	***573**	573	0	0.049	2
g50-4-01	460	***460**	460	0	0.387	*460	461.5	1.538	0.765	*460	460.3	0.923	0.542	3
g50-4-02	421	***421**	421	0	0.028	***421**	421	0	0.13	***421**	421	0	0.028	3
g50-4-03	565	***565**	565	0	0.154	***565**	565	0	0.477	***565**	565	0	0.26	3
g50-4-04	434	***434**	434	0	0.016	***434**	434	0	0.118	***434**	434	0	0.002	3
g50-4-05	387 (391)	***387**	387	0	0.058	***387**	387	0	0.216	***387**	387	0	0.058	3
g75-4-01	366	***366**	366	0	0.154	***366**	366	0	0.179	***366**	366	0	0.02	4
g75-4-02	295	***295**	295	0	0.394	***295**	295	0	0.21	***295**	295	0	0.016	4
g75-4-03	412 (421)	***412**	412	0	0.648	*412	412.399	0.502	1.776	***412**	412	0	0.013	4
g75-4-04	430 (432)	***430**	430	0	0.566	*430	430.1	0.307	1.385	***430**	430	0	0.141	4
g75-4-05	284	***284**	284	0	0.016	***284**	284	0	0.125	***284**	284	0	0.01	4
g100-4-01	363	*363	363.25	1.118	1.048	***363**	363	0	0.266	***363**	363	0	0.134	5
g100-4-02	335 (338)	*335	335.449	1.099	1.077	*335	335.149	0.67	1.084	***335**	335	0	0.128	5
g100-4-03	412	***412**	412	0	0.014	***412**	412	0	0.143	***412**	412	0	0.02	5
g100-4-04	442	***442**	442	0	0.018	***442**	442	0	0.172	***442**	442	0	0.063	5
g100-4-05	388	***388**	388	0	0.219	*388	388.449	2.012	1.485	***388**	388	0	0.164	5
g200-4-01	308	***308**	308	0	0.074	***308**	308	0	0.365	***308**	308	0	0.057	10
g200-4-02	299	***299**	299	0	0.068	***299**	299	0	0.424	***299**	299	0	0.156	10
g200-4-03	300	***300**	300	0	0.038	***300**	300	0	0.247	***300**	300	0	0.029	10
g200-4-04	304 (310)	***304**	304	0	0.719	***304**	304	0	1.058	***304**	304	0	0.304	10
g200-4-05	357	***357**	357	0	0.047	***357**	357	0	0.262	***357**	357	0	0.026	10
g400-4-01	253	***253**	253	0	0.075	***253**	253	0	0.27	***253**	253	0	0.021	20
g400-4-02	328 (338)	*328	328.699	3.13	3.998	***328**	328	0	2.502	***328**	328	0	0.191	20
g400-4-03	302	***302**	302	0	1.663	*302	302.05	0.223	4.322	***302**	302	0	0.231	20
g400-4-04	306 (314)	***306**	306	0	0.624	***306**	306	0	1.056	***306**	306	0	0.088	20
g400-4-05	320	***320**	320	0	1.84	***320**	320	0	2.387	***320**	320	0	0.303	20
g1000-4-01	263 (270)	*263	263.899	1.97	6.415	*263	263.699	1.838	5.293	***263**	263.5	1.235	8.423	20
g1000-4-02	281 (292)	*281	281.3	1.341	2.837	*281	281.3	1.341	6.716	***281**	281	0	1.036	20
g1000-4-03	289 (295)	***289**	289	0	1.445	*289	290.8	5.54	6.357	***289**	289	0	0.87	20
g1000-4-04	298 (306)	***298**	298	0	3.334	*298	305.6	1.788	3.367	***298**	298	0	6.291	20
g1000-4-05	268 (280)	***268**	268	0	0.391	***268**	268	0	1.292	***268**	268	0	0.072	20

Column 1 contains the instance name. The second column contains the best known solution value (if it was improved by our algorithms then the old best known solution is given in brackets). Then, there are 4 columns for each of our three algorithms. The first column gives the best found solutions in 20 runs (an asterisk denotes that it is equal to the (new) best known solution, and bold font indicates that the result is better or equal to the result of the other two approaches), whereas the second column gives the average of the best solutions in 20 runs. The standard deviation of this average is given in column 3, and the average computation time that was needed to find the best solution in a run is given in column 4.

6.1. A new set of benchmark instances

Regarding the lack of suitable problem instances, we decided to initiate a diverse and challenging set of benchmark instances. First of all we randomly generated 10 grid graph instances of varying size, because, as discovered in earlier papers (see for example [25]), the KCT problem seems to be especially difficult to solve in grid graph instances. Furthermore, we intended to incorporate problem instances of different edge-density and different variance of vertex degree. Considering the existence of many graph-based combinatorial optimization problems, we decided to borrow graphs from benchmark instance sets for other problems. We chose one of the Leighton graphs from the graph coloring benchmark set available from the OR-Library [38]. This graph is characterized by a high variance in vertex degrees, which is one of the indicators for the graph being clustered. We also chose 5 different graphs from the Steiner tree benchmark set that are also available from the OR-Library. This is justified by the fact that the Steiner tree problem is also a problem where trees are sought in a graph. Additionally, as mentioned above, we chose 4 of the biggest problem instances that were generated by Blesa and Xhafa using the software tool by Jörnsten and Løkketangen [21]. For all the graphs (except the 4 last mentioned, which already had assigned edge weights) we generated edge weights uniformly at random from the interval $[1, \dots, 100]$. An overview on the new set of benchmark instances is given in Table 3. The column with the heading $\bar{\Delta}$ contains the average vertex degrees of the graphs and the column with the heading $\sigma^2(\Delta)$ contains the variance of the vertex degrees. Our benchmark set can be obtained from [39].

Table 3
The new set of benchmark problem instances

Graph	Type	$\lvert V\rvert$	$\lvert E\rvert$	$\bar{\Delta}$	$\sigma^2(\Delta)$	Origin
bb15x15_1.gg	Grid graph	225	420	3.73	0.48	New
bb15x15_2.gg	Grid graph	225	420	3.73	0.48	New
bb45x5_1.gg	Grid graph	225	400	3.55	0.53	New
bb45x5_2.gg	Grid graph	225	400	3.55	0.53	New
bb33x33_1.gg	Grid graph	1089	2112	3.87	0.33	New
bb33x33_2.gg	Grid graph	1089	2112	3.87	0.33	New
bb100x10_1.gg	Grid graph	1000	1890	3.78	0.42	New
bb100x10_2.gg	Grid graph	1000	1890	3.78	0.42	New
bb50x50_1.gg	Grid graph	2500	4900	3.92	0.27	New
bb50x50_2.gg	Grid graph	2500	4900	3.92	0.27	New
g400-4-01.g	4-Regular graph	400	800	4.00	0.00	KCT problem [26]
g400-4-05.g	4-Regular graph	400	800	4.00	0.00	KCT problem [26]
g1000-4-01.g	4-Regular graph	1000	2000	4.00	0.00	KCT problem [26]
g1000-4-05.g	4-Regular graph	1000	2000	4.00	0.00	KCT problem [26]
steinc5.g	Sparse graph	500	625	2.5	1.65	Steiner tree problem
steind5.g	Sparse graph	1000	1250	2.5	1.57	Steiner tree problem
steine5.g	Sparse graph	2500	3125	2.5	1.57	Steiner tree problem
steinc15.g	Dense graph	500	2500	10.0	3.08	Steiner tree problem
steind15.g	Dense graph	1000	5000	10.0	3.22	Steiner tree problem
le450_15a.g	Dense graph	450	8168	36.30	16.83	Leighton graph, graph coloring

6.1.1. Cardinalities

We applied our algorithms to each problem instance for several cardinalities from the range of possible cardinalities. As minimal and maximal cardinality we chose 2, respectively $|V| - 2$. Note, that for cardinality 1 the problem is trivial, and for cardinality $|V| - 1$ the problem is equivalent to the minimum spanning tree problem which is polynomially solvable for example by Prims' algorithm [40]. From the range between the minimum and maximum cardinality we chose for each instance about 10 equidistantly distributed cardinalities. This was done in order to be able to study the possibly changing performance of the algorithms over the whole range of cardinalities.

6.1.2. Time limits

For the KCT problem it is a non-trivial task to find reasonable time limits for the algorithms, because they do not only depend on the number of edges and the number of nodes of the graph, but also on the cardinality k. In order to avoid artificial time limits we generated them for our algorithms as follows. We applied an enhanced version of the heuristic method K-CardPrim that was proposed in [16] to each combination of benchmark instance and cardinality. This heuristic works as follows: Starting from each node of the graph, a k-cardinality tree is constructed by applying a truncated version of Prims' algorithm [40]. To each of these k-cardinality trees we applied our steepest descent local search method (based on the neighborhood $\mathcal{N}_{\text{leaf}}$ as outlined in Section 2). As time limits for our metaheuristics we then chose for each combination of instance and cardinality 5 times the time that was needed by K-CardPrim. However, especially for bigger graphs, this time limit would have been impractically high. This was also due to the fact that we wanted to run our algorithms 20 times for every combination of instance and cardinality in order to increase the statistical significance of our results. Therefore, we have set the time limit for instances bb50x50_1.gg, bb50x50_2.gg, steine5.g, steind15.g and le450_15a.g for all cardinalities to the time that was needed by K-CardPrim (instead of 5 times that time). The results and the computation times of the heuristic K-CardPrim can be found at [39], as well as in an extended version of this paper [41].

6.2. Results and comparison

The amount of computation time that was needed to conduct all our experiments was more than 9 months of CPU time. Due to space limitations we present the numerical results that we obtained in a summarized way.[4] Figs. 1 and 2 show the relative behavior of our algorithms in comparison over the range of cardinalities.

First, we explain how to read the graphs that are shown in Fig. 2, which shows the average behavior of our algorithms over 20 runs (henceforth denoted by average-performance), and in Fig. 1, which shows the behavior of our algorithms in terms of the best solution they found in 20 runs (henceforth denoted by best-performance). In the 5 plots of Fig. 1, as well as in the 5 plots of Fig. 2, the x-axis shows the relative cardinalities, which are obtained by mapping the absolute cardinalities to the interval $(0, 1)$. For example, cardinality 10 for an instance with $|V| = 100$ is mapped to $\frac{10}{100} = 0.1$. This is done in order to be able to merge the results that our algorithms obtained on several instances of different size. The y-axis shows the performance of an algorithm in relation to the performance of the other two algorithms for a relative cardinality on a set of instances.

[4] The numerical results are presented in an extended version of the paper [41], as well as on the KCTLIB [39].

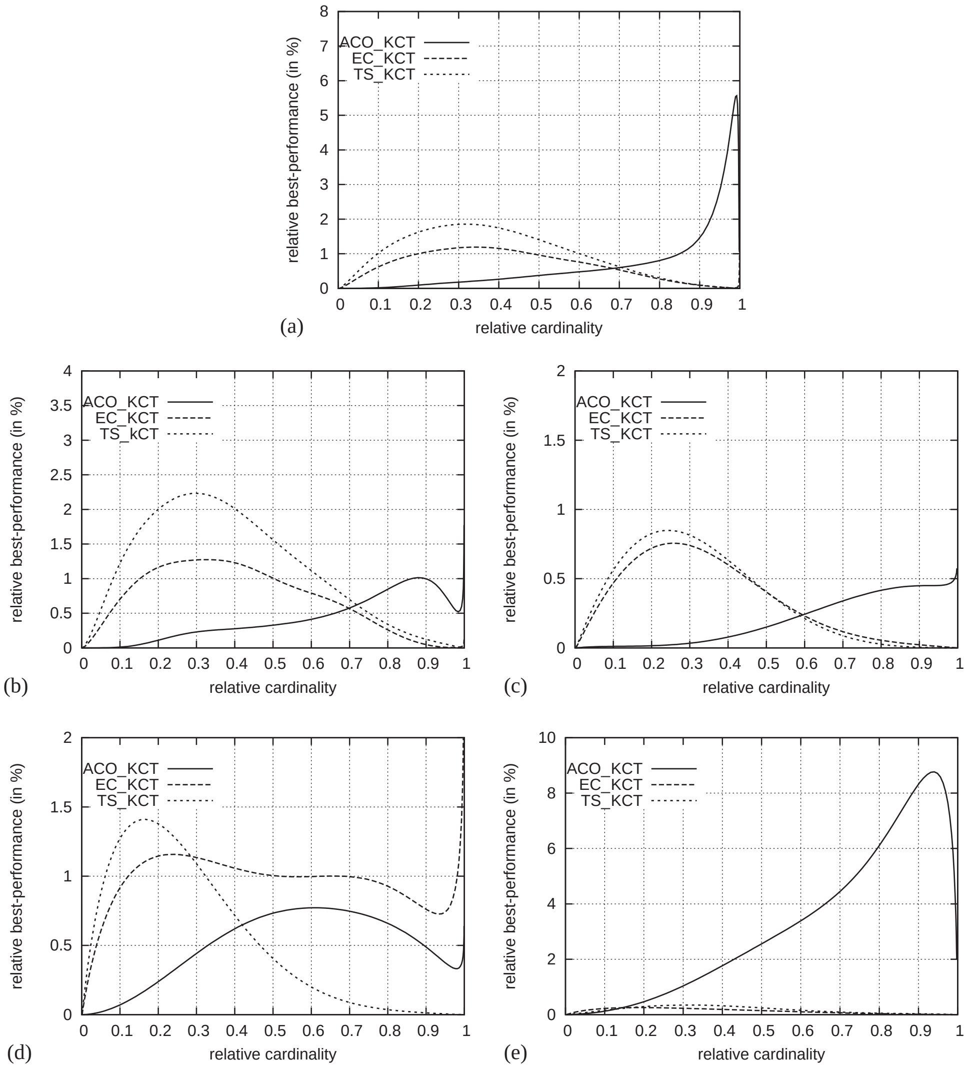

Fig. 1. Comparison of the three algorithms in terms of best-performance. The five plots show the relative performance of the three algorithms over the range of relative cardinalities for different subsets of our benchmark instances (as indicated by the subfigure labels). In general, the lower a curve, the better the corresponding algorithm. For a more detailed explanation see the beginning of Section 6.2. (a) Summary of all results; (b) summary of the results for grid graphs; (c) summary of the results for regular graphs; (d) summary of the results for sparse graphs; (e) summary of the results for dense graphs.

Consider for example the case in which for the relative cardinality 0.1 ACO_KCT has a y-value of 0.4, EC_KCT has a y-value of 1.2 and TS_KCT has a y-value of 1.8. This means that for the set of considered problem instances, for relative cardinality 0.1 the performance of ACO_KCT was on average 0.4 percent above the best algorithm performance, whereas the performance of EC_KCT was on average 1.2 percent above the best algorithm performance, and so on. In short, the lower a curve, the better.

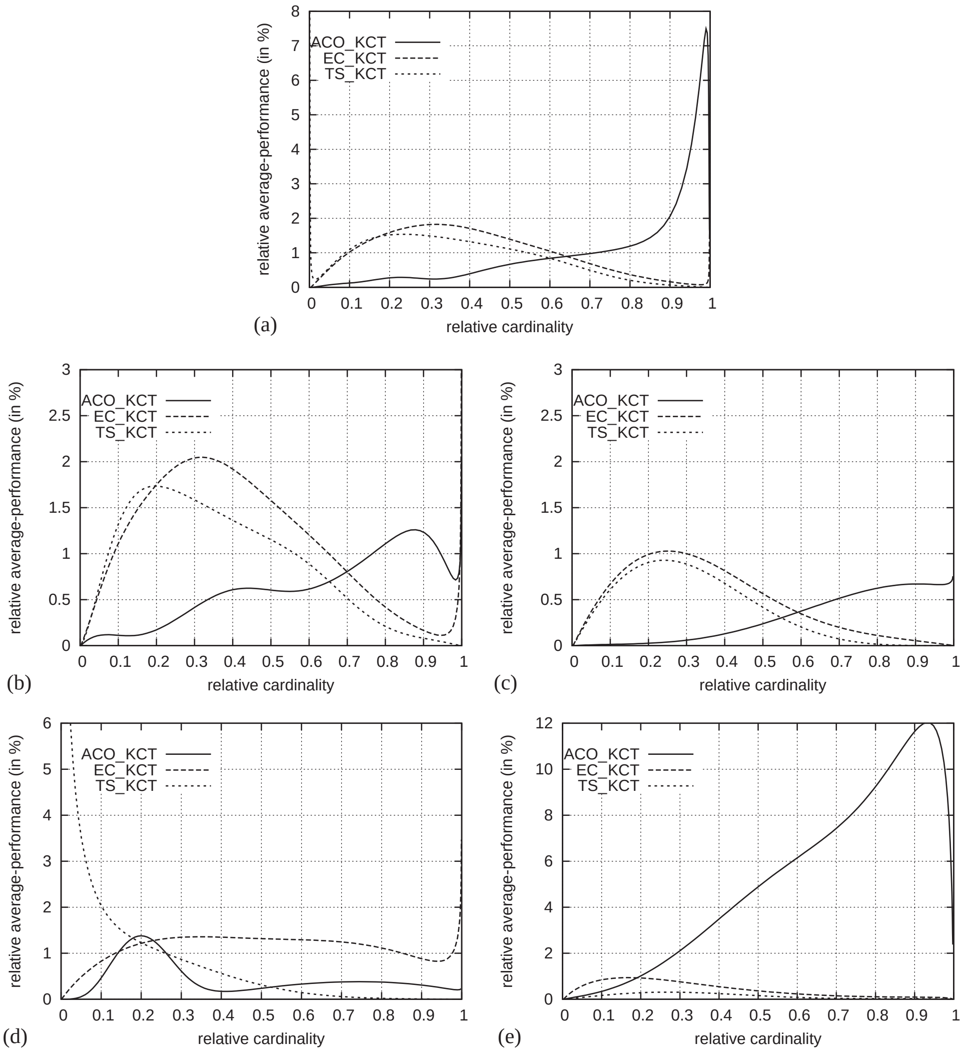

Fig. 2. Comparison of the three algorithms in terms of average-performance. The five plots show the relative performance of the three algorithms over the range of relative cardinalities for different subsets of our benchmark instances (as indicated by the subfigure labels). In general, the lower a curve, the better the corresponding algorithm. For a more detailed explanation see the beginning of Section 6.2. (a) Summary of all results; (b) summary of the results for grid graphs; (c) summary of the results for regular graphs; (d) summary of the results for sparse graphs; (e) summary of the results for dense graphs.

Figs. 1(a) and 2(a) show the relative performance of our algorithms when merged over all considered problem instances. We can observe, that in general all our algorithms are quite close together. They are in general within 2% in terms of best-performance as well as in terms of average-performance, except for ACO_KCT that decreases in performance for the high end of the cardinality range. However, for about the first 60% of the cardinality range ACO_KCT is in terms of both, best-performance as well as average-performance, the best algorithm. Both, EC_KCT and

TS_KCT, are inferior to ACO_KCT in the first 60% of the cardinality range. However, they show clear advantages in the last 40% of the cardinality range, namely for bigger cardinalities. It is interesting to note that EC_KCT seems to have a slight advantage over TS_KCT in terms of best-performance (as can be seen in Fig. 1(a)), whereas TS_KCT seems to have slight advantages over EC_KCT in terms of average-performance (as can be seen in Fig. 2(a)). This indicates the robustness of TS_KCT.

In order to study in more detail the relative performance of our algorithms on subclasses of our set of benchmark instances, we applied the same kind of analysis for subsets of our set of benchmark instances. We identified the following 4 subsets of instances with different characteristics:

- *Subset* 1: The 10 new grid graphs (see Figs. 1(b) and 2(b));
- *Subset* 2: The 4 4-regular graphs from the test set at [26] (see Figs. 1(c) and 2(c));
- *Subset* 3: The 3 sparse graphs steinc5.g, steind5.g and steine5.g (see Figs. 1(d) and 2(d));
- *Subset* 4: The 3 dense graphs steinc15.g, steind15.g and le450_15a.g (see Figs. 1(e) and 2(e));

Results on subsets 1 *and* 2. On the grid graphs and the 4-regular graphs ACO_KCT has even stronger advantages over the other two approaches for the first 60%–70% of the cardinality range. And even though ACO_KCT is inferior to the other two approaches for the rest of the cardinalities, the performance is just slightly worse (on average not more than 1%). On the 4-regular graphs all three approaches are over the whole cardinality range in terms of best-performance as well as in terms of average-performance quite close together (within about 1%).

Results on subset 3. The results for sparse graphs give a different impression of the relative performance of our three approaches. TS_KCT is in both performance measures (best-performance as well as average-performance) the best algorithm for the second half of the cardinality range. In the first half of the cardinality range, TS_KCT has slight disadvantages compared to the other two approaches in terms of best-performance. Even more, the average-performance of TS_KCT in the first half of the cardinality range indicates that it is not very robust for smaller cardinalities in sparse graphs. This can be explained as follows. As the neighborhood $\mathcal{N}_{\text{leaf}}$ only allows the removal of leaf edges from a k-cardinality tree, a "bad" edge that has a relatively high distance to leaf edges can only be removed from the tree by basically leaving the current area of the search space. So, even if TS_KCT manages to get rid of such a bad edge, it is difficult to find the way back to this area of the search space, as many of the edges from this area will be in the tabu list. This problem is not as apparent in dense graphs, because there are many more edges available. When comparing ACO_KCT with EC_KCT on sparse graphs, we note that the ACO approach is consistently better than the EC approach in both performance measures, except for the average-performance at about 20% into the cardinality range, where the ACO approach seems to have some difficulties.

Results on subset 4. When comparing the three approaches on dense graphs it immediately becomes clear that ACO_KCT has difficulties there and is clearly outperformed by EC_KCT and TS_KCT over the whole cardinality range. The reason for that might be, that because of the high number of edges in relation to the number of nodes the convergence speed is much lower and more computation time might be required in order to reach good solutions. In terms of best-performance EC_KCT and TS_KCT are very close together. However, in terms of average-performance TS_KCT outperforms EC_KCT. Also, the computation times needed by TS_KCT in order to find its best solutions are much lower than the computation times needed by EC_KCT (see [41,39]). The fact that TS_KCT appears to be the best approach for dense graphs confirms our explanation for the weakness of TS_KCT on sparse graphs.

Summary. Our results show that the characteristics of a problem instance as well as the size of the cardinality have a high influence on the behavior of our three approaches. This leads to the fact that none of our metaheuristic approaches can be identified as being overall the best one. It is rather the case that different metaheuristic approaches have advantages for certain classes of problem instances (e.g., TS_KCT is the best approach for dense graphs), or certain ranges of cardinalities (e.g., ACO_KCT is generally the best approach for small to medium size cardinalities). This outcome is very much as expected, as for most combinatorial optimization problems where different classes of problem instances are available it does not exist "one" best metaheuristic approach. As an example we mention the Quadratic Assignment Problem (QAP), which was extensively studied and tackled by many researchers. For random instances TS approaches are generally best, whereas for structured instances metaheuristics such as ACO, EC, or Iterated Local Search (ILS) reach best performance.

7. Conclusions and outlook

We proposed three different metaheuristic approaches to tackle the edge-weighted k-cardinality tree (KCT) problem. These are a Tabu Search approach that is characterized by a dynamic tabu list length for balancing intensification and diversification, a new Evolutionary Computation approach based on two heuristically guided crossover operators, and an Ant Colony Optimization approach. All our approaches use a simple neighborhood structure that is efficient to compute. The Tabu Search uses this neighborhood structure for performing moves, whereas the Evolutionary Computation and the Ant Colony Optimization approach use this neighborhood structure in terms of black-box local search procedures for improving solutions. First, we showed the competitiveness of our algorithms by applying them to existing benchmark instances and comparing the results to existing approaches. Then, we developed a diverse set of benchmark instances containing graphs from several different classes of graphs, such as grid graphs, regular graphs, dense graphs and sparse graphs. Finally, we conducted a large amount of experiments and presented the results in graphical as well as in numerical form. The results showed that the performance of our metaheuristic approaches is largely determined by the graph class and the cardinality. This result is not surprising as this is the case for many other hard combinatorial optimization problems such as for example the Quadratic Assignment Problem. In the future it might be interesting to explore the use of more expensive neighborhood structures for the local search procedures. Finally, we invite other researchers to implement their algorithms on the same basic data structures in order to improve comparability with our results. We also welcome the proposal of additional benchmark instances such as random graphs, geometric graphs or small-world graphs.

Acknowledgements

Christian Blum acknowledges support by the "Metaheuristics Network", a Research Training Network funded by the Improving Human Potential program of the CEC, Grant HPRN-CT-1999-00106. The information provided is the sole responsibility of the authors and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication. Maria J. Blesa acknowledges support by the Catalan Research Council of the

Generalitat de Catalunya (Grant 2001FI-00659), by the FET Programme of the EU under Contract Nos. IST-1999-14186 (ALCOM-FT) and IST-2001-33116 (FLAGS) and by the Spanish CICYT Projects TIC-2000-1970-CE, TIC-2001-4917-E and TIC-2002-04498-C05-03 (TRACER).

References

[1] Hamacher HW, Jörnsten K, Maffioli F. Weighted k-cardinality trees. Technical Report 91.023, Politecnico di Milano, Dipartimento di Elettronica, Italy, 1991.

[2] Hamacher HW, Jörnsten K. Optimal relinquishment according to the Norwegian petrol law: a combinatorial optimization approach. Technical Report No. 7/93, Norwegian School of Economics and Business Administration, Bergen, Norway, 1993.

[3] Foulds LR, Hamacher HW, Wilson J. Integer programming approaches to facilities layout models with forbidden areas. Annals of Operations Research 1998;81:405–17.

[4] Foulds LR, Hamacher HW. A new integer programming approach to (restricted) facilities layout problems allowing flexible facility shapes. Technical Report 1992–3, University of Waikato, Department of Management Science, 1992.

[5] Philpott AB, Wormald NC. On the optimal extraction of ore from an open-cast mine. New Zealand: University of Auckland; 1997.

[6] Borndörfer R, Ferreira C, Martin A. Matrix decomposition by branch-and-cut. Technical Report, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, 1997.

[7] Borndörfer R, Ferreira C, Martin A. Decomposing matrices into blocks. SIAM Journal on Optimization 1998;9(1): 236–69.

[8] Cheung SY, Kumar A. Efficient quorumcast routing algorithms. In: Proceedings of INFOCOM'94. Los Alamitos, USA, Silver Spring, MD: IEEE Society Press; 1994.

[9] Garg N, Hochbaum D. An $O(\log k)$ approximation algorithm for the k minimum spanning tree problem in the plane. Algorithmica 1997;18(1):111–21.

[10] Fischetti M, Hamacher HW, Jörnsten K, Maffioli F. Weighted k-cardinality trees: complexity and polyhedral structure. Networks 1994;24:11–21.

[11] Marathe MV, Ravi R, Ravi SS, Rosenkrantz DJ, Sundaram R. Spanning trees short or small. SIAM Journal on Discrete Mathematics 1996;9(2):178–200.

[12] Maffioli F. Finding a best subtree of a tree. Technical Report 91.041, Politecnico di Milano, Dipartimento di Elettronica, Italy, 1991.

[13] A. Zelikovsky, D. Lozevanu, Minimal and bounded trees. In: Tezele Cong. XVIII Acad. Romano-Americane, Kishinev, 1993. p. 25–6.

[14] Freitag J. Minimal k-cardinality trees. Master's thesis, Department of Mathematics, University of Kaiserslautern, Germany, 1993 [in German].

[15] Uehara R. The number of connected components in graphs and its applications. IEICE Technical Report COMP99-10, Natural Science Faculty, Komazawa University, Japan, 1999.

[16] Ehrgott M, Freitag J, Hamacher HW, Maffioli F. Heuristics for the k-cardinality tree and subgraph problem. Asia-Pacific Journal of Operational Research 1997;14(1):87–114.

[17] Ehrgott M, Freitag J. K_TREE/K_SUBGRAPH: a program package for minimal weighted k-cardinality-trees and -subgraphs. European Journal of Operational Research 1996;1(93):214–25.

[18] Blum C, Roli A. Metaheuristics in combinatorial optimization: overview and conceptual comparison. ACM Computing Surveys 2003;35(3):268–308.

[19] Blesa MJ, Moscato P, Xhafa F. A memetic algorithm for the minimum weighted k-cardinality tree subgraph problem. In: Proceedings of the Metaheuristics International Conference MIC'2001, vol. 1, Porto, Portugal, July 2001. p. 85–90.

[20] Blesa MJ, Xhafa F. A C++ implementation of tabu search for k-cardinality tree problem based on generic programming and component Reuse. In: Net.ObjectDays 2000 Tagungsband, Erfurt, Germany, 2000. p. 648–52. Net.ObjectDays-Forum.

[21] Jörnsten K, Løkketangen A. Tabu search for weighted k-cardinality trees. Asia-Pacific Journal of Operational Research 1997;14(2):9–26.

[22] Løkketangen A. Tabu search—using the search experience to guide the search process. An introduction with examples. AICOM 1995;8(2):78–85.

[23] Mladenović N, Urošević D. Variable neighborhood search for the k-cardinality tree problem. In: Proceedings of the Metaheuristics International Conference MIC'2001, vol. 2, 2001. p. 743–7.

[24] Blum C. Ant colony optimization for the edge-weighted k-cardinality tree problem. In: Langdon WB, Cantú-Paz E, Mathias K, Roy R, Davis D, Poli R, Balakrishnan K, Honavar V, Rudolph G, Wegener J, Bull L, Potter MA, Schultz AC, Miller JF, Burke E, Jonoska N, editors. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002). San Mateo, CA: Morgan Kaufmann Publishers; 2002. p. 27–34.

[25] Blum C, Ehrgott M. Local search algorithms for the k-cardinality tree problem. Discrete Applied Mathematics 2003;128:511–40.

[26] MALLBA-Project. www.lsi.upc.es/∼mallba/public/library/TabuSearch/minktree.html, 2001.

[27] Glover F. Future paths for integer programming and links to artificial intelligence. Computers and Operations Research 1986;5:533–49.

[28] Glover F, Laguna M. Tabu search. Dordrecht: Kluwer Academic Publishers; 1997.

[29] Bäck T. Evolutionary algorithms in theory and practice. New York: Oxford University Press; 1996.

[30] Fogel DB. An introduction to simulated evolutionary optimization. IEEE Transactions on Neural Networks 1994;5(1):3–14.

[31] Hertz A, Kobler D. A framework for the description of evolutionary algorithms. European Journal of Operational Research 2000;126:1–12.

[32] Dorigo M. Optimization, learning and natural algorithms. Ph.D. thesis, DEI, Politecnico di Milano, Italy, 1992, 140pp [in Italian].

[33] Dorigo M, Di Caro G, Gambardella LM. Ant algorithms for discrete optimization. Artificial Life 1999;5(2):137–72.

[34] Dorigo M, Maniezzo V, Colorni A. Ant system: optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man and Cybernetics—Part B 1996;26(1):29–41.

[35] Dorigo M, Gambardella LM. Ant Colony System: a cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation 1997;1(1):53–66.

[36] Stützle T, Hoos HH. $\mathcal{MAX}$–$\mathcal{MIN}$ ant system. Future Generation Computer Systems 2000;16(8):889–914.

[37] Blum C, Roli A, Dorigo M. HC-ACO: the hyper-cube framework for ant colony optimization. In: Proceedings of Metaheuristics International Conference MIC'2001. Porto, Portugal, July 2001. p. 399–403.

[38] Beasley JE. OR-library: distributing test problems by electronic mail. Journal of the Operational Research Society 1990;41(11):1069–72.

[39] KCTLIB. http://iridia.ulb.ac.be/∼cblum/kctlib/, 2003.

[40] Cormen T, Leisersoon C, Rivest R. Introduction to algorithms 2nd ed. Cambridge, MA: MIT Press; and New York: McGraw-Hill; 2001.

[41] Blum C, Blesa MJ. Metaheuristics for the edge-weighted k-cardinality tree problem. Technical Report TR/IRIDIA/2003-02, IRIDIA, Université Libre de Bruxelles, Belgium, 2003. Also available as technical report LSI-03-1-R, LSI, Universitat Politècnica de Catalunya, Spain.