Combining ant colony optimization with dynamic programming for solving the k-cardinality tree problem

Christian Blum
Maria J. Blesa

Report LSI–05–18–R

April 22, 2005

Departament de Llenguatges i Sistemes Informatics
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Combining ant colony optimization with dynamic programming for solving the k-cardinality tree problem[⋆]

Christian Blum and Maria Blesa

ALBCOM, Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya, E-08034 Barcelona, Spain
{cblum,mjblesa}@lsi.upc.edu

Abstract. Research efforts in metaheuristics have shown that an intelligent incorporation of more classical optimization techniques in metaheuristics can be very beneficial. In this paper, we combine the metaheuristic ant colony optimization with dynamic programming for the application to the NP-hard k-cardinality tree problem. Given an undirected graph G with node and/or edge weights, the problem consists of finding a tree in G with exactly k edges such that the sum of the weights is minimal. In a standard ant colony optimization algorithm, ants construct trees with exactly k edges. In our algorithm, ants may construct trees that have more than k edges, in which case we use a recent dynamic programming algorithm to find—in polynomial time—the best k-cardinality tree embedded in the bigger tree constructed by the ants. We show that our hybrid algorithm improves over the standard ant colony optimization algorithm and, for node-weighted grid graph instances, is a current state-of-the-art method.

1 Introduction

The k-cardinality tree (KCT) problem—also referred to as the *k-minimum spanning tree* (k-MST) problem, or just the *k-tree* problem—is an NP-hard [13] combinatorial optimization problem which generalizes the well-known minimum weight spanning tree problem. In this paper we deal with a generalized problem version in which the given graph G can have both node and edge weights. More formally, let $G = (V, E)$ be a graph with a weight function $w_E : E \rightarrow \mathbb{N}$ on the edges and a weight function $w_V : V \rightarrow \mathbb{N}$ on the nodes. We denote by $\mathcal{T}_k$ the set of all k-cardinality trees (i.e., trees with exactly k edges) in G. Then, the problem consists of finding a k-cardinality tree $T_k \in \mathcal{T}_k$ that minimizes

$$f(T_k) = \left(\sum_{e \in E(T_k)} w_E(e) \right) + \left(\sum_{v \in V(T_k)} w_V(v) \right) \quad , \tag{1}$$

[⋆] This work was supported by the Spanish CICYT project no. TIC-2002-04498-C05-03 (TRACER), and by the "Juan de la Cierva" program of the Spanish Ministry of Science and Technology of which Christian Blum is a post-doctoral research fellow.

where $E(T)$ and $V(T)$ denote the edges of the tree T and its nodes, respectively.

The edge-weighted version of the KCT problem was first tackled by exact approaches [14, 8, 17] and heuristics [12, 11, 8]. Soon, the research focused on the development of more appealing metaheuristics: two evolutionary computation approaches [1, 4], three tabu search methods [2, 15, 4], different variations of variable neighborhood search (VNS) [18] and two ant colony optimization (ACO) approaches [7, 4]. Two sets of benchmark instances exist: one was introduced for the empirical evaluation of the VNS-based approaches in [18], and the other one for the metaheuristics proposed in [4]. The variable neighborhood decomposition search (VNDS) algorithm proposed in [18] is the state-of-the-art method for the first set, and the ACO algorithm proposed in [7] is so for the second set.

Less results are known for the node-weighted KCT problem. Greedy-based heuristics were proposed in [12], and the first metaheuristic approaches were presented in [5]. The only existing benchmark set for the node-weighted KCT, together with the currently best metaheuristic (a VNDS), are introduced in [6].

Our contribution. A polynomial-time dynamic programming (DP) algorithm for finding optimal k-cardinality trees in bigger edge-weighted trees was proposed in [16]. This algorithm was used in well-working heuristics for the edge-weighted KCT problem [12]. Recently, we extended this algorithm to be applied to the general (edge and/or node-weighted) KCT problem [3]. We show how a standard ACO algorithm (see [4]) can be improved by applying the DP algorithm in [3] in the following way: Instead of producing k-cardinality trees, the ants produce trees with more than k edges. To these trees we then apply the dynamic programming algorithm in order to obtain the best k-cardinality trees embedded in them. Our experimental results show that our algorithm improves over the standard ACO algorithm for the KCT problem. Moreover, our algorithm is able to improve current state-of-the-art results for node-weighted grid graph instances.

The remainder of the paper is organized as follows. In Section, 2 we describe our hybrid algorithm. In Section 3, we present the experimental evaluation, before we conclude this work in Section 4.

2 The ACO-DP algorithm

ACO [9, 10] emerged in the early 90's as a novel nature-inspired metaheuristic for the solution of combinatorial optimization problems. The inspiring source of ACO is the foraging behaviour of real ants. When searching for food, ants initially explore the area surrounding their nest in a random manner. As soon as an ant finds a food source, it carries some of the food found to the nest. During the return trip, the ant deposits a chemical pheromone trail on the ground. The quantity of pheromone deposited, which may depend on the quantity and quality of the food, will guide other ants to the food source. The indirect communication established via the pheromone trails allows the ants to find shortest paths between their nest and food sources. This behaviour of real ants in nature is exploited in ACO in order to solve discrete optimization problems using artificial ant colonies.

Algorithm 1 ACO-DP for the KCT problem

```
INPUT: a node and/or edge-weighted graph $G$, a cardinality $k < |V| - 1$, and
a tree size $l$ with $k \leq l \leq |V| - 1$
$T_k^{bs} \leftarrow$ NULL, $T_k^{rb} \leftarrow$ NULL
$cf \leftarrow 0$, $bs\_update \leftarrow$ FALSE
forall $e \in E$ do $\tau_e \leftarrow 0.5$ end forall
while termination conditions not satisfied do
  for $j = 1$ to $n_a$ do
    ${T_l}^j \leftarrow$ ConstructTree($\mathcal{T}$,$l$)
    if ($l > k$) then ${T_k}^j \leftarrow$ DynamicTree(${T_l}^j$) end if
  end for
  $T_k^{ib} \leftarrow$ argmin$\{f({T_k}^1), \ldots, f({T_k}^{n_a})\}$
  Update($T_k^{ib}$,$T_k^{rb}$,$T_k^{bs}$)
  ApplyPheromoneValueUpdate($cf$,$bs\_update$,$\mathcal{T}$,$T_k^{ib}$,$T_k^{rb}$,$T_k^{bs}$)
  $cf \leftarrow$ ComputeConvergenceFactor($\mathcal{T}$, $T_k^{rb}$)
  if $cf \geq 0.99$ then
    if $bs\_update$ = TRUE then
      forall $e \in E$ do $\tau_e \leftarrow 0.5$ end forall
      $T_k^{rb} \leftarrow$ NULL
      $bs\_update \leftarrow$ FALSE
    else
      $bs\_update \leftarrow$ TRUE
    end if
  end if
end while
OUTPUT: $T_k^{bs}$
```

We combine a standard ACO algorithm [4] with the dynamic programming algorithm in [3], and denote the resulting algorithm as ACO-DP. Algorithm 1 captures the framework of this new hybrid approach. In the pheromone model used, the set of pheromone trail parameters $\mathcal{T}$ contains a parameter $\mathcal{T}_e$ (with pheromone value τ_e) for each $e \in E$. After the initialization of the variables T_k^{bs} (i.e., the best-so-far solution), T_k^{rb} (i.e., the restart-best solution), and *cf* (i.e., the convergence factor), all the pheromone values are set to 0.5. At every iteration, each of the n_a ants construct probabilistically an l-cardinality tree. If $l > k$, the dynamic programming algorithm from [3] is applied to extract the best k-cardinality tree embedded in the l-cardinality tree. Finally, before the next iteration starts, some of the solutions are used for updating the pheromone values. The details of the methods in this framework are explained in the following.

– ConstructTree($\mathcal{T}$,l): Henceforth, we denote the two nodes that are connected by an edge e with $v_{e,1}$ and $v_{e,2}$. To build an l-cardinality tree, an ant starts from an edge e that is chosen probabilistically in proportion to the values $\tau_e/(w_E(e) + w_V(v_{e,1}) + w_V(v_{e,2}))$. At each construction step an ant extends its current tree by adding a node and an edge such that the result is again a tree.

Let, at an arbitrary construction step, $\mathcal{N}$ be the set of nodes that fulfill the above condition. For each $v \in \mathcal{N}$ let $\mathcal{N}_v$ be the set of edges that have v as an end-point, ant that have their other end-point—denoted by $v_{e,o}$—in the current tree. If an ant chooses $v \in \mathcal{N}$ to be added to the current tree, edge $e_{\min} \in \mathcal{N}_v$ that minimizes $w_{_E}(e) + w_{_V}(v_{e,o})$ is also added to the current tree. It remains to be specified how an ant chooses a node $v \in \mathcal{N}$: In $det\,\%$ of the cases, an ant chooses the node v that minimizes $w_{_E}(e_{\min}) + w_{_V}(v)$. In $100 - det\,\%$ of the cases, v is chosen probabilistically in proportion to $w_{_E}(e_{\min}) + w_{_V}(v)$.

– DynamicTree($T_l{}^j$): This procedure applies the dynamic programming algorithm for node and/or edge-weighted trees in [3] to an l-cardinality tree $T_l{}^j$ (where j denotes the tree constructed by the jth ant). The algorithm returns the best k-cardinality tree $T_k{}^j$ embedded in $T_l{}^j$.

– Update(T_k^{ib},T_k^{rb},T_k^{bs}): In this procedure T_k^{rb} and T_k^{bs} are set to T_k^{ib} (i.e., the iteration-best solution), if $f(T_k^{ib}) < f(T_k^{rb})$ and $f(T_k^{ib}) < f(T_k^{bs})$, respectively.

– ApplyPheromoneUpdate(*cf*,*bs_update*,$\mathcal{T}$,T_k^{ib},T_k^{rb},T_k^{bs}): In the same way as described in [4], our ACO-DP algorithm may use three different solutions for updating the pheromone values: (i) the iteration-best solution T_k^{ib}, (ii) the restart-best solution T_k^{rb} and, (iii) the best-so-far solution T_k^{bs}. Their influence depends on the convergence factor cf, which provides an estimate about the state of convergence of the system. To perform the update, first an update value ξ_e for every pheromone trail parameter $\mathcal{T}_e \in \mathcal{T}$ is computed:

$$\xi_e \leftarrow \kappa_{ib} \cdot \delta(T_k^{ib}, e) + \kappa_{rb} \cdot \delta(T_k^{rb}, e) + \kappa_{bs} \cdot \delta(T_k^{bs}, e) \quad ,$$

where κ_{ib} is the weight of T_k^{ib}, κ_{rb} the weight of T_k^{rb}, and κ_{bs} the weight of T_k^{bs} such that $\kappa_{ib} + \kappa_{rb} + \kappa_{bs} = 1.0$. The δ-function is the characteristic function of the set of edges in the tree, i.e., for each k-cardinality tree T_k,

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

Then, the following update rule is applied to all pheromone values τ_e :

$$\tau_e \leftarrow \min\left\{\max\{\tau_{\min}, \tau_e + \rho \cdot (\xi_e - \tau_e)\}, \tau_{\max}\right\} \quad ,$$

where $\rho \in (0, 1]$ is the evaporation (or learning) rate. The upper and lower bounds $\tau_{\max} = 0.99$ and $\tau_{\min} = 0.01$ keep the pheromone values always in the range $(\tau_{\min}, \tau_{\max})$, thus preventing the algorithm from converging to a solution. After tuning, the values for ρ, κ_{ib}, κ_{rb} and κ_{bs} are chosen as shown in Table 1.

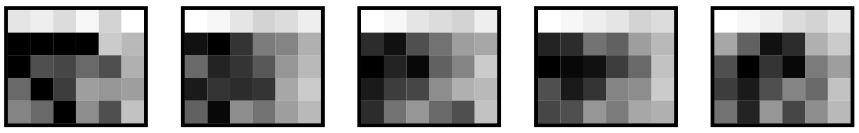
– ComputeConvergenceFactor($\mathcal{T}$, T_k^{rb}): This function computes, at each iteration, the convergence factor as

$$cf \leftarrow \frac{\sum_{e \in E(T_k^{rb})} \tau_e}{k \cdot \tau_{\max}} \quad ,$$

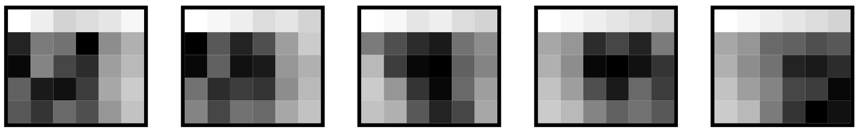
where $\tau_{\max}$ is again the upper limit for the pheromone values. The convergence factor cf can therefore only assume values between 0 and 1. The closer cf is to 1, the higher is the probability to produce the solution T_k^{rb}.

Table 1. The schedule used for values ρ, κ_{ib}, κ_{rb} and κ_{bs} depending on cf (the convergence factor) and the Boolean control variable *bs_update*.

	$bs\_update = \text{FALSE}$			$bs\_update = \text{TRUE}$
	$cf < 0.7$	$cf \in [0.7, 0.9)$	$cf \geq 0.9$	
ρ	0.05	0.1	0.15	0.15
κ_{ib}	2/3	1/3	0	0
κ_{rb}	1/3	2/3	1	0
κ_{bs}	0	0	0	1



(a) Tuning results for 20x20 random node-weighted grid graphs. From left to right, the results for cardinalities 40, 80, 120, 160, and 200 are shown.



(b) Tuning results for edge-weighted graphs with 500 nodes. From left to right, the results for cardinalities 50, 100, 150, 200, and 250 are shown.

Fig. 1. Tuning results. In each of the five matrices in (a) and (b), the columns correspond to the 6 values of *det*, and the rows to the 5 values of l (e.g., the matrix position $(2, 2)$ corresponds to the gray value of the tuple $(det = 80, l = k + s)$).

3 Experimental results

We implemented ACO-DP in C++, and run experiments on a PC with a 3 GHz Intel Pentium IV processor and 1 Gb memory. In the previous section we specified the values of all parameters of ACO-DP except for two crucial ones: *det*, the percentage of deterministic steps during the tree construction, and l, the size of the trees constructed by the ants. For their determination, we conducted a parameter tuning by running ACO-DP for all combinations of $det \in \{75, 80, 85, 90, 95, 99\}$ and $l \in \{k, k+s, k+2s, k+3s, k+4s\}$, where $s = (|V| - 1 - k)/4$.

To obtain instances for tuning, we randomly generated 10 20x20-grid node-weighted graph instances, and applied ACO-DP exactly once for each (det, l) combination and for each cardinality $k \in \{40, 80, 120, 160, 200\}$. This gives a value averaged over the 10 graphs for each (det, l, k) triple. For each k, we then ranked the resulting 30 values and translated them into gray scale: the best of the 30 values received gray value 1.0 (i.e., black), and the worst received gray value

Table 2. Results for node-weighted grid graphs in [6]. As time limits for ACO-DP we used one seventh of the time limits of VNDS as given in [6].

n	k	VNDS	DynamicTree (Prim)	ACO-DP		
		q_{vnds}	q_{dtp}	q_{aco}	$d_{vnds}/\%$	$d_{dtp}/\%$
30x30	100	8571.9	8612.2	**8203.7**	-4.2954	-4.7432
	200	17994.4	18491.6	**17850.1**	-0.8019	-3.4691
	300	**28770.9**	29488.6	28883.9	0.3927	-2.0506
	400	**42114**	42618.6	42331.9	0.5174	-0.6727
	500	**59266.4**	59662	59541.7	0.4645	-0.2016
40x40	150	18029.9	18495.3	**17527.1**	-2.7887	-5.2348
	300	38965.9	39220.4	**37623.8**	-3.4442	-4.0708
	450	61290.1	62118.3	**60417**	-1.4245	-2.7388
	600	**86422.3**	87935.4	86594.7	0.1994	-1.5246
	750	**117654**	119303	118570	0.7785	-0.6140
50x50	250	37004	38007.1	**35995.2**	-2.7261	-5.2934
	500	81065.8	80247.9	**77309.9**	-4.6331	-3.6611
	750	128200	128224	**125415**	-2.1723	-2.1908
	1000	182220	184103	**181983**	-0.1300	-1.1516
	1250	**250962**	253116	253059	0.8355	-0.0224
$\bar{d}/\%$					-1.2818	-2.5093

0.0 (i.e., white). These results, expressed in gray values, are shown in Figure 1(a). We did the same for the 10 edge-weighted graphs with 500 nodes from the benchmark set proposed in [18] (for cardinalities $k \in \{50, 100, 150, 200, 250\}$). The results are shown in Figure 1(b).

Both for node-weighted grid graphs and for edge-weighted graphs, the tuning results show that a setting $l > k$ is always better, which means that ACO-DP always improves on the standard ACO algorithm. With respect to the tuning results, we chose the setting (det = 85, l=k+2s) for the application of ACO-DP to node-weighted grid graphs, and the setting ($det = 95, l = k + 2s$) for the application to edge-weighted graphs. With these settings, we applied our algorithm to some of the existing benchmarks.[1]

Results for the node-weighted benchmark. We applied ACO-DP to all node-weighted grid graph instances from [6], i.e., 10 30x30 instances, another 10 of 40x40, and another 10 with 50x50 nodes. The results are shown in Table 2. The first table column indicates the graph type, and the second one indicates the tested cardinality. We compared our results to the state-of-the-art algorithm VNDS in [6] and to the heuristic DynamicTree (Prim) in [3]; the latter constructs a spanning tree of the given graph, and applies the dynamic programming algorithm to it. For these approaches, together with our ACO-DP approach, the table shows the average value obtained for each (n, k) combination (note that each algorithm was applied exactly once to each of the 10 instances of a graph type). The best value for each (n, k) combination is given in bold font. For ACO-

[1] Concerning the computational overhead, note that with $l = k + 2s$ the time needed for constructing a solution for our smallest cardinality takes about 6 times longer than with $l = k$. While for our biggest cardinality the time needed for constructing a solution is only about 0.3 times longer than with $l = k$.

Table 3. Results for edge-weighted graphs in [18]. As time limits for ACO-DP we used one tenth of the time limits of VNDS as given in [18].

n	k	VNDS	DynamicTree (Prim)	ACO-DP		
		q_{vnds}	q_{dtp}	q_{aco}	$d_{vnds}/\%$	$d_{dtp}/\%$
1000	100	5828	5841.9	**5827.2**	-0.0137	-0.2516
	200	**11893.7**	11927.5	11910.1	0.1378	-0.1458
	300	**18196.6**	18225.6	18217.4	0.1143	-0.0449
	400	**24734**	24766.4	24757.8	0.0962	-0.0347
	500	**31561.8**	31593.3	31613.8	0.1647	0.0648
2000	200	23538.3	23543.6	**23479**	-0.2519	-0.2743
	400	**48027.3**	48086.2	48030.4	0.0064	-0.1160
	600	**73277.8**	73394.2	73392.9	0.1570	-0.0017
	800	**99491.2**	99623.2	99801.4	0.3117	0.1788
	1000	**126485**	126916	127325	0.6639	0.3222
3000	300	35186.1	35203.5	**35160.3**	-0.0733	-0.1227
	600	**71634.7**	71729.2	71862.8	0.3184	0.1862
	900	**109463**	109631	110094	0.5764	0.4223
	1200	**148826**	148997	149839	0.6803	0.5649
	1500	**189943**	190080	191627	0.8867	0.8141
$\overline{d}/\%$					0.2516	0.1041

DP we additionally provide the relative deviations (in %) from VNDS (headed by $d_{vnds}/\%$), and from DynamicTree (Prim) (headed by $d_{dtp}/\%$). The last table row provides the averages over these relative deviations. The machine used for running VNDS is about 7 times slower than our machine. Therefore, we used as time limits for ACO-DP one seventh of the time limits of VNDS in [6].

The results show that ACO-DP is on average 1.28 % better than VNDS, and 2.51 % better than DynamicTree (Prim). They also show that ACO-DP seems in general to have advantages for smaller cardinalities, and that ACO-DP seems to become better—in comparison to VNDS—with growing problem instance size.

Results for the edge-weighted benchmark. We also applied ACO-DP to some of the edge-weighted graph instances from [18], i.e., 10 instances with 1000, 2000 and 3000 nodes, respectively. The results are shown in Table 3 in the same way as outlined for Table 2. Note that also a VNDS (a different one) is the current state-of-the-art algorithm for these instances. The machine that was used to run VNDS is about 10 times slower than our machine. Therefore, we used as time limits for ACO-DP one tenth of the time limits of VNDS in [18].

The results show that in contrast to the node-weighted grid graph case, ACO-DP seems not to reach the state-of-the-art results; ACO-DP can improve them only for small cardinalities. This might be due to the much higher density of the edge-weighted graph instances (w.r.t. the node-weighted grid graph instances). Note that the difference in quality is minimal; on average ACO-DP is only 0.25 % worse than VNDS, and 0.1 % worse than DynamicTree (Prim).

4 Conclusions

A successful combination of ACO and dynamic programming has been proposed for solving a general (edge and/or node-weighted) version of the NP-hard k-

cardinality tree problem. The results obtained indicate that the hybrid algorithm outperforms the standard ACO algorithm. Furthermore, for node-weighted grid graphs, the hybrid algorithm outperforms the current state-of-the-art VNDS method. Further experiments have shown that the results obtained depend on topological properties of the graphs (e.g., density, d-regularity, etc.) rather than the tackled problem version (i.e., edge or node weighted).

References

1. M. Blesa, P. Moscato, and F. Xhafa. A memetic algorithm for the minimum weighted k-cardinality tree subgraph problem. In *Proc. of the 4th Metah. Intern. Conf.*, volume 1, pages 85–90, 2001.
2. M. Blesa and F. Xhafa. A C++ implementation of tabu search for k-cardinality tree problem based on generic programming and component reuse. In *Net.ObjectDays 2000 Tagungsband*, pages 648–652, Erfurt, Germany, 2000.
3. C. Blum. Revisiting dynamic programming for finding optimal subtrees in trees. Tech. Rep. LSI-04-57, Universitat Politècnica Catalunya, Spain, 2004. Submitted.
4. C. Blum and M. Blesa. New metaheuristic approaches for the edge-weighted k-cardinality tree problem. *Computers & Operations Research*, 32:1355–1377, 2005.
5. C. Blum and M. Ehrgott. Local search algorithms for the k-cardinality tree problem. *Discr. Appl. Math.*, 128:511–540, 2003.
6. J. Brimberg, D. Urošević, and N. Mladenović. Variable neighborhood search for the vertex weighted k-cardinality tree problem. *Eur. J. of Op. Res.*, 2005. In press.
7. T. N. Bui and G. Sundarraj. Ant system for the k-cardinality tree problem. In K. Deb et al., editor, *Proc. of the Genetic and Ev. Comp. Conference–GECCO 2004*, volume 3102 of LNCS, pages 36–47. Springer, 2004.
8. S. Y. Cheung and A. Kumar. Efficient quorumcast routing algorithms. In *Proceedings of INFOCOM'94*, Los Alamitos, USA, 1994. IEEE Society Press.
9. M. Dorigo. *Optimization, Learning and Natural Algorithms* (in Italian). PhD, Dipartimento di Elettronica, Politecnico di Milano, Italy, 1992.
10. M. Dorigo, V. Maniezzo, and A. Colorni. Ant System: Optimization by a colony of cooperating agents. *IEEE TSMC – Part B*, 26:29–41, 1996.
11. M. Ehrgott and J. Freitag. K_TREE / K_SUBGRAPH: A program package for minimal weighted k-cardinality-trees and -subgraphs. *Eur. J. of. Op. Res.*, 1:214–225, 1996.
12. M. Ehrgott, J. Freitag, H. W. Hamacher, and F. Maffioli. Heuristics for the k-cardinality tree and subgraph problem. *Asia-Pac. J. of Op. Res.*, 14:87–114, 1997.
13. M. Fischetti, H. W. Hamacher, K. Jørnsten, and F. Maffioli. Weighted k-cardinality trees: Complexity and polyhedral structure. *Networks*, 24:11–21, 1994.
14. J. Freitag. *Minimal k-cardinality trees* (in German). Master's thesis, Department of Mathematics, University of Kaiserslautern, Germany, 1993.
15. K. Jörnsten and A. Løkketangen. Tabu search for weighted k-cardinality trees. *Asia-Pac. J. of Op. Res*, 14:9–26, 1997.
16. F. Maffioli. Finding a best subtree of a tree. Tech. Rep. 91.041, Politecnico di Milano, Dipartimento di Elettronica, Italy, 1991.
17. R. Uehara. The number of connected components in graphs and its applications. IEICE Tech. Rep. COMP99-10, Komazawa University, Japan, 1999.
18. D. Urošević, J. Brimberg, and N. Mladenović. Variable neighborhood decomposition search for the edge weighted k-cardinality tree problem. *Comp. & Op. Res.*, 31:1205–1213, 2004.