A new hybrid evolutionary algorithm for the k-cardinality tree problem[*]

Christian Blum

ALBCOM, LSI, Universitat Politècnica de Catalunya
Jordi Girona 1-3, Campus Nord, 08034 Barcelona, Spain
cblum@lsi.upc.es

Abstract

In recent years it has been shown that an intelligent combination of metaheuristics with other optimization techniques can significantly improve over the application of a pure metaheuristic. In this paper, we combine the evolutionary computation paradigm with dynamic programming for the application to the NP-hard k-cardinality tree problem. Given an undirected graph G with node and edge weights, this problem consists of finding a tree in G with exactly k edges such that the sum of the weights is minimal. The genetic operators of our algorithm are based on an existing dynamic programming algorithm from the literature for finding optimal subtrees in a given tree. The simulation results show that our algorithm is able to improve the best known results for benchmark problems from the literature in 111 cases.

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 [14, 23] combinatorial optimization problem which generalizes the well-known minimum weight spanning tree problem. Let $G = (V, E)$ be a graph with a weight function $w_E : E \to \mathbb{N}$ on the edges and a weight function $w_V : V \to \mathbb{N}$ on the nodes. We denote the weight of a node v by $w_V(v)$ (or just w_v), and the weight of an edge e by $w_E(e)$ (or just w_e). Furthermore, we denote by $\mathcal{T}_k$ the set of all k-cardinality trees in G, that is, the set of all trees in G with exactly k edges. Then, the problem consists of finding a k-cardinality tree $T_k \in \mathcal{T}_k$ that minimizes

$$f(T_k) = \left(\sum_{e \in E_{T_k}} w_e \right) + \left(\sum_{v \in V_{T_k}} w_v \right) \quad . \tag{1}$$

In this equation, as well as in the rest of the paper, when given a tree T, E_T denotes the set of edges of T, and V_T the set of nodes of T.

The KCT problem was first described in [20] and it has gained considerable interest since the mid 1990's due to various applications, e.g. in oil-field leasing [19], facility layout

[*]This work was supported by the Spanish CICYT project OPLINK (grant TIN-2005-08818-C04-01), 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.

[15, 16], open pit mining [24], matrix decomposition [7, 8], quorum-cast routing [11] and telecommunications [18].

The edge weighted version of the KCT problem (i.e., node weights are all zero) was first tackled by exact approaches [17, 11, 25] and heuristics [13, 12, 11]. The best ones of these heuristics are based on a polynomial time dynamic programming algorithm [22, 2] that finds the best k-cardinality tree in a graph that is itself a tree. However, the interest in heuristics was quickly lost and research focused on the development of more appealing metaheuristics [6]. Among these, the different versions of variable neighborhood search (VNS) proposed in [26] can be regarded as state-of-the-art for the benchmark instance set proposed in the same paper, and the ant colony optimization (ACO) approach proposed in [10] is currently state-of-the-art for the benchmark instance set proposed in [4]. Much less research efforts were directed at the node weighted KCT problem. Simple greedy as well as dual greedy based heuristics were proposed in [13], and the first metaheuristic approaches were presented in [5]. The best technique for the node-weighted KCT problem is the variable neighborhood descent (VNDS) technique proposed in [9]. In the same paper the only existing benchmark set for the node weighted KCT was introduced.

Motivation for this paper In [2] we extended the dynamic programming algorithm of Maffioli, which was introduced for edge-weighted trees, to trees that can have both edge and node weights. In the same article we conducted an experimental evaluation of two simple heuristics for the KCT problem in graphs with node and/or edge weights. Both heuristics are based on this extended dynamic programming algorithm. The results concerning (almost) all available benchmark instances for the edge weighted and for the node weighted KCT problem showed that the current state-of-the-art metaheuristics are on average only slightly better than these two heuristics, while consuming much more computation time. Therefore, we advocated the hybridization of this dynamic programming algorithm with metaheuristics. In [3] we made such an attempt by developing a hybrid between ACO and dynamic programming. The results show that for node weighted instances and rather small cardinalities this hybrid algorithm improves on the results of the VNDS algorithm proposed in [13]. However, the results for edge weighted problem instances were inferior to the results of the VNS algorithm proposed in [26]. In this paper we make a different use of the dynamic programming algorithm. More specifically, we propose an evolutionary algorithm whose genetic operators are based on dynamic programming.

The organization of this paper is as follows. In Section 2 we outline the hybrid evolutionary algorithm. Extensive computational tests of this algorithm are presented in Section 3, and Section 4 offers conclusions and an outlook to the future.

2 Hybrid EA for the KCT problem

Evolutionary algorithms (EAs) [1, 21] are widely used to tackle hard optimization problems. They are inspired by nature's capability to evolve living beings which are well adapted to their environment. EAs can shortly be characterized as computational models of evolutionary processes working on populations of individuals. Individuals are in most cases solutions to the tackled problem. EAs apply genetic operators such as *recombination* and/or *mutation* operators in order to generate new solutions at each iteration. The driving force in EAs 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 EAs.

2.1 Tree construction

The operators of our hybrid EA are based on the principle of tree construction. A tree construction is well-defined by the definition of the following four components:

1. The graph $G' = (V', E')$ in which the tree should be constructed;
2. The number $l \leq (V' - 1)$ of edges of the tree to be constructed (also called the size of the tree);
3. The way in which to start the tree construction (e.g., by determining a node or an edge from which to start the construction process);
4. The way in which to perform each of the construction steps.

The first three definitions are operator dependent. For example, in the operator for generating the initial population a tree construction might be started from a randomly chosen edge, whereas the crossover operator might start a tree construction from a partial tree. However, the way in which to perform a construction step is the same in all algorithm operators: Given a graph $G' = (V', E')$, the desired size l of the final tree, and the current tree T whose size is smaller than l, a construction step consists of adding exactly one node and one edge to T such that the result is again a tree. Let, at an arbitrary construction step, $\mathcal{N}$, with $\mathcal{N} \cup V_T = \emptyset$, be the set of nodes of G' that can be added to T via at least one edge.[1] For each $v \in \mathcal{N}$ let E_v be the set of edges that have v as an end-point, and that have their other end-point—denoted by $v_{e,o}$—in T. Then, a node $v \in \mathcal{N}$ is chosen as follows. With probability $\mathbf{p}_{det}$, v is chosen as the node that minimizes $w_{e_{\min}} + w_v$. Hereby,

$$e_{\min} \leftarrow \operatorname{argmin}\{w_e + w_{v_{e,o}} \mid e \in E_v\} \ . \tag{2}$$

Otherwise (i.e., with probability $1 - \mathbf{p}_{det}$), v is chosen probabilistically in proportion to $w_{e_{\min}} + w_v$. This means that when $\mathbf{p}_{det}$ is close to 1, the tree construction is almost deterministic, and the other way around. The way in which we chose a value for this parameter is outlined in Section 3. Finally, to complete the tree construction step, v and $e_{\min}$ are added to T. For an example see Figure 1.

2.2 The algorithm

The algorithmic framework of our hybrid EA approach to tackle the KCT problem is shown in Algorithm 1. In this algorithm, henceforth denoted by HyEA, T_k^{best} denotes the best solution (i.e., the best k-cardinality tree) found since the start of the algorithm, and T_k^{iter} denotes the best solution in the current population P. The algorithm starts by generating the initial population in function GenerateInitial- Population(pop_size). Then, at each iteration the algorithm produces a new population by first applying a crossover operator in function ApplyCrossover(P), and then by the subsequent replacement of the worst solutions with newly

[1] Remember that V_T denotes the node set of T.

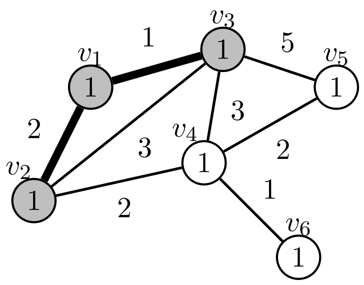

Figure 1: In this example we have given a graph with 6 nodes and 8 edges. The nodes weights are for simplicity reasons all set to 1. Nodes and edges are labelled with their weights. Furthermore we have given a tree T of size 2, denoted by gray shaded nodes and bold edges: $V_T = \{v_1, v_2, v_3\}$, and $E_T = \{e_{1,2}, e_{1,3}\}$. The set of nodes that can be added to T is therefore given as $\mathcal{N} = \{v_4, v_5\}$. The set of edges that join v_4 with T is $E_{v_4} = \{e_{2,4}, e_{3,4}\}$, and the set of edges that join v_5 with T is $E_{v_5} = \{e_{3,5}\}$. Due to the edge weights, $e_{\min}$ is in the case of v_4 determined as $e_{2,4}$, and in the case of v_5 as $e_{3,5}$.

Algorithm 1 Hybrid EA for the KCT problem (HyEA)

```
INPUT: a node and/or edge-weighted graph $G$, and a cardinality $k < |V| - 1$
$P \leftarrow$ GenerateInitialPopulation($pop\_size$)
${T_k}^{best} \leftarrow \text{argmin}\{f(T_k) \mid T_k \in P\}$
while termination conditions are not met do
    $\hat{P} \leftarrow$ ApplyCrossover($P$)
    $P \leftarrow$ IntroduceNewMaterial($\hat{P}$)
    ${T_k}^{iter} \leftarrow \text{argmin}\{f(T_k) \mid T_k \in P\}$
    if $f({T_k}^{iter}) < f({T_k}^{best})$ then
        ${T_k}^{best} \leftarrow {T_k}^{iter}$
    end if
end while
OUTPUT: ${T_k}^{best}$
```

generated trees in function IntroduceNewMaterial($\hat{P}$). The components of this algorithm are outlined in more detail below.

GenerateInitialPopulation(pop_size): The initial population is generated in this method. It takes as input the size *pop_size* of the population. We explain in Section 3 how *pop_size* is determined. The construction of each of the initial k-cardinality trees in graph G starts with a node that is chosen uniformly at random from V. All further construction steps are performed as described in Section 2.1.

ApplyCrossover(P): At each algorithm iteration an offspring population $\hat{P}$ is generated from the current population P. For each k-cardinality tree $T \in P$, the following is done. First, tournament selection (with tournament size 3) is used to choose a crossover parter $T^c \neq T$ for T from P. In the following we say that two trees in the same graph are overlapping, if and only if they have at least one node in common.

In case T and T^c are overlapping, graph G^c is defined as the union of T and T^c, that is, $V_{G^c} = V_T \cup V_{T^c}$ and $E_{G^c} = E_T \cup E_{T^c}$. Then, a spanning tree T^{sp} of G^c is constructed as follows. The first node is chosen uniformly at random. Each further construction step is performed as described in Section 2.1. Then the dynamic programming algorithm proposed in [2] is applied to T^{sp} for finding the best k-cardinality tree T^{child} that is contained in T^{sp}.

Otherwise, that is, in case the crossover partners T and T^c are not overlapping, T is used as the basis for constructing a tree in G that contains both, T and T^c. This is done by extending T (with construction steps as outlined in Section 2.1) until the current tree under construction can be connected with T^c by at least one edge. In case of several connecting edges, edge e that minimizes $w_e + w_{v_a} + w_{v_b}$ is chosen, where v_a and v_b are the two endpoints of e. Finally, we apply the dynamic programming algorithm proposed in [2] for finding the best k-cardinality tree T^{child} in the constructed tree.

The better tree among T^{child} and T is added to the offspring population $\hat{P}$.

IntroduceNewMaterial($\hat{P}$): In order to avoid a premature convergence of the algorithm, this function introduces at each iteration new material (in the form of newly constructed k-cardinality trees) into the population. The input of this function is the offspring population $\hat{P}$ generated by crossover. First, the function selects $X = \lfloor 100 - newmat \rfloor\%$ of the best solutions in $\hat{P}$ for the new population P. Then, the remaining $100 - X\%$ of P are generated as follows: Starting from a node of G that is uniformly chosen at random, an l_{new}-cardinality tree $T_{l_{new}}$ (where $l_{new} \geq k$) is constructed by applying construction steps as outlined in Section 2.1. In Section 3 we describe the setting of l_{new}. Then, the dynamic programming algorithm proposed in [2] is used for finding the best k-cardinality tree in $T_{l_{new}}$. This tree is then added to P.

Our hybrid EA algorithm HyEA outputs the best solution found during a run. This completes the description of the algorithm.

3 Experimental evaluation

We implemented HyEA in ANSI C++ using GCC 3.2.2 for compiling the software. Our experimental results were obtained on a PC with Intel Pentium 4 processor (3.06 GHz) and 1 Gb of memory. Before we present the computational results, we specify in the following the setting of the four algorithm parameters.

3.1 Setting of the algorithm parameters

***pop_size*:** The population size is one of the important algorithm parameters. Earlier experience with the KCT problem (see, for example, [4]) has shown that the population size should be coupled to the cardinality k. That is, when k is rather small, the population size should be rather big, and vice versa. Based on some initial computational tests we decided for a population size *pop_size* such that each node of G (on average) appears in 5 members of the population. In addition, we introduced sensible lower and upper bounds for the population size (i.e., at least 10 population members, respectively at most 100 population members). In technical terms,

$$pop\_size \leftarrow \min\left\{\max\left\{10, \left\lfloor 5 \cdot \frac{|V|}{k+1} \right\rfloor\right\}, 100\right\} \quad . \tag{3}$$

$\mathbf{p}_{det}$: As outlined in Section 2.1, when constructing a tree each step may either be performed deterministically or probabilistically. This is decided for each construction step with a certain probability $\mathbf{p}_{det}$. If $\mathbf{p}_{det}$ is close to one, a tree construction is almost deterministic, and if $\mathbf{p}_{det}$ is close to zero, the tree construction is almost probabilistic. By means of experimentation we found that for rather small cardinalities a value of about $\mathbf{p}_{det} = 0.85$ works well, whereas for rather big cardinalities a value around $\mathbf{p}_{det} = 0.99$ works best. In order to obtain an algorithm that shows a reasonable performance over the whole cardinality range, we decided to chose $\mathbf{p}_{det}$ for each tree construction (or tree extension in the case of the crossover operator) uniformly at random from $[0.85, 0.99]$.

***newmat*:** This parameter takes an integer value between 0 and 100, and denotes the percentage of new trees in the new population that is generated per algorithm iteration. Clearly, if $newmat = 0$ the algorithm will suffer from premature convergence, whereas if $newmat = 100$, the algorithm will just create a random new population at each iteration. After tuning by hand we found the value of $newmat = 20$ to be reasonably well working for small as well as big cardinalities. However, this parameter is not really critical. Values between 10 and 30 work almost equally well.

l_{new}: At each iteration, $newmat\%$ new trees are added to the new population. These trees might be generated in the same way as the trees of the initial population, that is, trees of size k (corresponding to a setting of $l_{new} = k$). However, we noticed that—in particular in later stages of the search process—the quality of these k-cardinality trees was not comparable to the quality of the trees resulting from the evolution process. Adding these trees was therefore not useful for the search process. Hence, we decided to generate trees that are bigger than k (i.e., $l_{new} > k$), and to apply the dynamic programming algorithm proposed in [2] in order to find the best k-cardinality tree in the l_{new}-cardinality tree that was generated. There is of course a trade-off between time and quality. Generating trees of size $l_{new} = |V| - 1$ (i.e., spanning trees) results on average in the best k-cardinality trees. However, in this case the dynamic programming algorithm takes more time than in the case of $l_{new} < |V| - 1$. After some experimentation we decided for a value of

$$l_{new} \leftarrow k + \left\lfloor \frac{|V| - 1 - k}{3} \right\rfloor \ . \tag{4}$$

In words, l_{new} is set to k plus one third of the remaining cardinality range (remember that the maximum cardinality is $|V| - 1$).

3.2 Results

3.2.1 Application to the edge-weighted instances (part 1)

First, we applied our algorithm to 12 of the edge-weighted graphs from the benchmark set by Blum and Blesa [4]; that is, the same 12 instances to which the current state-of-the-art algorithm for this benchmark set—an ant colony optimization approach (denoted by ACO) by Bui and Sundarraj [10]—was applied. The results are shown in Tables 2, 3, and 4. The format of these tables is as follows. The first column provides the cardinality, while the second column contains the value of the best known solution for the respective cardinality. The cases in which the best known solution was improved by HyEA are marked by a left-right arrow. Columns 3 and 4 provide the value of the best solution found in 20 runs by ACO, respectively the average of the best solutions found in the 20 runs. The same information is given for HyEA

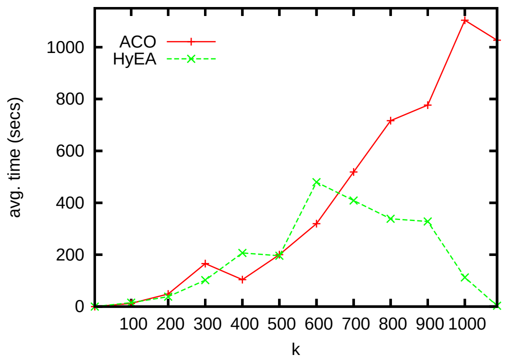

Figure 2: Average computation time (in seconds) of ACO and HyACO over the cardinality range $[2, 1087]$ of problem instance bb33x33_1.

in columns 5 and 6. Additionally, in column 7 is provided the average time needed to find the best solutions in the 20 runs. For space reasons this information is not provided for ACO. The time information can be obtained from [10]. We show the differences in computation time graphically on the typical example of problem instance bb33x33_1 in Figure 2. Hereby one has to keep in mind that the results of ACO were obtained on a computer with Intel Pentium 4 processor (2.4 GHz) and 512 Gb of memory. The graphic in Figure 2 shows that for small and medium size cardinalities the computation times of ACO and HyEA are comparable. However, for larger cardinalities HyEA has clear advantages over ACO in terms of computation time. As computation time limits for HyEA we used the time limits that were used in [4] (divided by 2.7, due to the fact that the machine used in [4] is about 2.7 times slower than the machine that we used).

When comparing the computational results displayed in Tables 2, 3, and 4, we note a clear advantage of HyEA over ACO. Table 1 shows a summary of the results that are provided in Tables2, 3, and 4. When small graphs are concerned the results of both methods are comparable. The advantage of HyEA over ACO is especially strong on bigger graphs such as, for example, g1000-4-01 or bb33x33_1. With respect to the feasible cardinality range, the advantage of HyEA over ACO is especially clear for smaller cardinalities. Altogether, HyEA improves the best known solutions concerning the considered 12 instances of this benchmark set in 53 cases.

Table 1: Summary of the results displayed in Tables 2 to 13. The numbers in the table show how often—in 138 results—HyEA is better than, respectively "worse than" or "equal to", ACO. This information is given with respect to the best solutions found (second table row), and the average results obtained (third table row).

	better	**equal**	**worse**
best	53	71	14
average	78	45	15

3.2.2 Application to the edge-weighted instances (part 2)

Second, we applied our algorithm to the 8 remaining edge-weighted graphs from the benchmark set by Blum and Blesa [4]. The only algorithms that were applied so far to these 8 instances were the algorithms proposed in [4]. As none of these approaches is clearly better than the others, we compare HyEA only to the best known solutions (see Table 5). Especially concerning the bigger ones of these graphs, HyEA has clear advantages over the best solutions found by the three approaches proposed in [4]. Alltogether, HyEA is able to improve 51 of the best known solutions.

3.2.3 Application to node-weighted instances

In a third set of experiments we applied HyEA to the benchmark set of node-weighted graphs that was proposed in [9]. This set is composed of 30 grid graphs, that is, 10 grid graphs of 900 vertices (i.e., 30 times 30 vertices), 10 grid graphs of 1600 vertices, and 10 grid graphs of 2500 vertices. Furthermore, the benchmark set consists of 30 random graphs, that is, 10 graphs of 3000 vertices, 10 graphs of 4000 vertices, and 10 graphs of 5000 vertices. We compared our results to the results of the variable neighborhood decent technique (denoted by VNDS) presented in [9], and to the results of the hybrid ant colony optimization technique (denoted by HyACO) that we proposed in [3]. This comparison is shown in Table 6. Instead of applying an algorithm several times to the same graph and cardinality, it is usual for this benchmark set to apply the algorithm exactly once to each graph and cardinality, and then to average the results over the graphs of the same type. Therefore, the structure of Table 6 is slightly different to the tables of the previous section. The first table column indicates the graph type (e.g., grid graphs of size 30x30). The second table column contains the cardinality, whereas the third table column provides the best known results (abbreviated by **bkr**). Then for each of the three algorithms we provide the result together with the average computation time that was spent in order to compute this result. Note that the computation time limit for HyEA was the same as the one that was used for HyACO (see [3]).

For what concerns the application to grid graph instances (see Table 6a), we note that both HyACO as well as HyEA are in 9 out of 15 cases better than VNDS. It is interesting to note that this concerns especially the cases of small to medium size cardinalities. For larger cardinalities VNDS beats both HyACO and HyEA. Furthermore, HyEA is in 11 out of 15 cases better than HyACO. This indicates that, even though HyACO and HyEA behaver similar in comparison to VNDS, HyEA seems to make a better use of the dynamic programming algorithm than HyACO. In terms of computation time, both algorithms are comparable.

Even though all three algorithms provide very similar results, VNDS seems to have a consistent advantage over HyEA and HyACO for what concerns the application to random graph instances (see Table 6b). When comparing HyEA with HyACO we note that HyEA is in 12 out of 15 cases better than HyACO. In fact, the three cases in which HyACO beats HyEA are the smallest cardinalities concerning the three graph types. This suggests that except for the application to very small cardinalities HyEA has in general advantages over HyACO when applied to node-weighted random graph instances.

4 Conclusions and outlook

In this paper we have proposed a hybrid evolutionary algorithm for the application to the k-cardinality tree problem, which is an NP-hard combinatorial optimization problem. The hybrid component of our algorithm is a dynamic programming algorithm for finding optimal trees in graphs that are themselfs trees. This dynamic programming algorithm is used in all the operators of our algorithm. We conducted an extensive computational evaluation of our algorithm. The results are especially favorable when edge-weighted problem instances are concerned. In fact, our algorithm is able to improve 104 best known solutions for benchmark instances from the literature. Furthermore, our algorithm is comparable to current state-of-the-art algorithms when applied to node-weighted grid graph instances. In 6 out of 15 cases our algorithm is able to improve the best known results from the literature. On the negative side, our algorithm seems to have some problems for node-weighted random graph instances. These instances are the only ones for which our algorithm does not reach state-of-the-art performance. We plan to investigate on this topic in future research.

References

[1] T. Bäck. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, New York, 1996.

[2] C. Blum. Revisiting dynamic programming for finding optimal subtrees in trees. *European Journal of Operational Research*, 2006. In press.

[3] C. Blum and M. Blesa. Combining Ant Colony Optimization with Dynamic Programming for solving the k-cardinality tree problem. In *8th International Work-Conference on Artificial Neural Networks, Computational Intelligence and Bioinspired Systems (IWANN'05)*, volume 3512 of *Lecture Notes in Computer Science*, pages 25–33. Springer-Verlag, Berlin, 2005.

[4] C. Blum and M.J. Blesa. New metaheuristic approaches for the edge-weighted k-cardinality tree problem. *Computers & Operations Research*, 32(6):1355–1377, 2005.

[5] C. Blum and M. Ehrgott. Local search algorithms for the k-cardinality tree problem. *Discrete Applied Mathematics*, 128:511–540, 2003. Selected for the Editors' choice volume of 2003.

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

[7] R. Borndörfer, C. Ferreira, and A. Martin. Matrix decomposition by Branch-and-Cut. Technical report, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, 1997.

[8] R. Borndörfer, C. Ferreira, and A. Martin. Decomposing matrices into blocks. *SIAM Journal on Optimization*, 9(1):236–269, 1998.

[9] J. Brimberg, D. Urošević, and N. Mladenović. Variable neighborhood search for the vertex weighted k-cardinality tree problem. *European Journal of Operational Research*, 2005. In press.

[10] T. N. Bui and G. Sundarraj. Ant system for the k-cardinality tree problem. In K. Deb et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference – GECCO 2004*, volume 3102 of *Lecture Notes in Computer Science*, pages 36–47. Springer Verlag, Berlin, Germany, 2004.

[11] S. Y. Cheung and A. Kumar. Efficient quorumcast routing algorithms. In *Proceedings of INFOCOM'94*, Los Alamitos, USA, 1994. IEEE Society Press.

[12] M. Ehrgott and J. Freitag. K_TREE / K_SUBGRAPH: A program package for minimal weighted k-cardinality-trees and -subgraphs. *European Journal of Operational Research*, 1(93):214–225, 1996.

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

[14] M. Fischetti, H. W. Hamacher, K. Jørnsten, and F. Maffioli. Weighted k-cardinality trees: Complexity and polyhedral structure. *Networks*, 24:11–21, 1994.

[15] L. R. Foulds and H. W. Hamacher. A new integer programming approach to (restricted) facilities layout problems allowing flexible facility shapes. Technical Report 1992-3, Department of Management Science, University of Waikato, New Zealand, 1992.

[16] L. R. Foulds, H. W. Hamacher, and J. Wilson. Integer programming approaches to facilities layout models with forbidden areas. *Annals of Operations Research*, 81:405–417, 1998.

[17] J. Freitag. Minimal k-cardinality trees. Master's thesis, Department of Mathematics, University of Kaiserslautern, Germany, 1993. In german.

[18] N. Garg and D. Hochbaum. An $O(\log k)$ approximation algorithm for the k minimum spanning tree problem in the plane. *Algorithmica*, 18:111–121, 1997.

[19] H. W. Hamacher and K. Joernsten. 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.

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

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

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

[23] M. V. Marathe, R. Ravi, S. S. Ravi, D. J. Rosenkrantz, and R. Sundaram. Spanning trees short or small. *SIAM Journal on Discrete Mathematics*, 9(2):178–200, 1996.

[24] H. W. Philpott and N. Wormald. On the optimal extraction of ore from an open-cast mine. Technical report, University of Auckland, New Zeland, 1997.

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

[26] D. Urošević, J. Brimberg, and N. Mladenović. Variable neighborhood decomposition search for the edge weighted k-cardinality tree problem. *Computers & Operations Research*, 31:1205–1213, 2004.

Table 2: Results for grid graphs bb15x15_1, bb15x15_2, bb33x33_1, and bb33x33_2.

(a) Grid graph bb15x15_1 (225 vertices)

k	**bks**	ACO		HyEA		
		best	**avg**	**best**	**avg**	**avg. time**
2	2	**2**	2.00	**2**	2.00	0.01
20	257	**257**	258.00	**257**	257.00	0.20
40	642	**642**	644.40	**642**	642.00	0.22
60	977	**977**	1005.50	**977**	978.50	0.39
80	1335	**1335**	1429.15	**1335**	1346.80	1.07
100	→ 1761	1762	1780.05	**1761**	1762.60	0.95
120	2235	**2235**	2262.80	**2235**	2235.00	0.77
140	2781	**2781**	2798.10	2783	2793.00	2.85
160	3417	**3417**	3423.00	**3417**	3424.30	2.41
180	4158	**4158**	4162.15	**4158**	4165.20	2.55
200	5040	**5040**	5040.95	5041	5041.00	0.54
220	6176	**6176**	6176.00	**6176**	6176.00	0.31
223	6400	**6400**	6400.00	**6400**	6400.00	0.01

(b) Grid graph bb15x15_2 (225 vertices)

k	**bks**	ACO		HyEA		
		best	**avg**	**best**	**avg**	**avg. time**
2	6	**6**	6.00	**6**	6.00	0.01
20	253	**253**	253.00	**253**	253.00	0.06
40	585	**585**	624.10	**585**	585.00	0.54
60	927	**927**	986.05	**927**	930.80	0.53
80	1290	**1290**	1348.35	1291	1295.00	1.01
100	1686	**1686**	1726.25	**1686**	1688.50	1.46
120	2120	**2120**	2143.55	**2120**	2124.20	1.47
140	2634	**2634**	2639.60	**2634**	2639.90	2.21
160	→ 3248	3250	3272.75	**3248**	3248.20	2.48
180	3915	**3915**	3915.00	**3915**	3915.00	0.74
200	4718	**4718**	4718.00	**4718**	4718.00	0.47
220	5862	**5862**	5862.00	**5862**	5862.00	0.03
223	6101	**6101**	6101.00	**6101**	6101.00	0.01

(c) Grid graph bb33x33_1 (1089 vertices)

k	**bks**	ACO		HyEA		
		best	**avg**	**best**	**avg**	**avg. time**
2	3	**3**	3.00	**3**	3.00	0.05
100	→ 1562	1587	1594.60	**1562**	1586.30	15.17
200	→ 3303	3366	3466.35	**3303**	3324.90	38.16
300	→ 5112	5235	5320.45	**5112**	5128.60	102.02
400	→ 7070	7166	7224.80	**7070**	7086.00	206.63
500	→ 9204	9256	9327.60	**9204**	9236.70	195.84
600	11579	**11579**	11579.00	11588	11607.60	479.65
700	→ 14299	14309	14313.35	**14299**	14311.10	408.93
800	→ 17393	17399	17399.00	**17393**	17405.50	338.28
900	→ 20919	20921	20921.00	**20919**	20920.10	328.29
1000	25199	**25199**	25199.00	**25199**	25199.00	112.54
1087	30417	**30417**	30417.00	**30417**	30417.00	3.27

(d) Grid graph bb33x33_2 (1089 vertices)

k	**bks**	ACO		HyEA		
		best	**avg**	**best**	**avg**	**avg. time**
2	3	**3**	3.00	**3**	3.00	0.09
100	1524	1531	1568.65	**1524**	1525.30	14.74
200	→ 3255	3316	3530.60	**3255**	3273.40	40.03
300	→ 5185	5275	5360.10	**5185**	5196.60	81.25
400	→ 7252	7340	7582.05	**7252**	7266.10	195.82
500	→ 9465	9514	9624.70	**9465**	9484.00	263.24
600	→ 11856	11879	11889.30	**11856**	11886.90	285.07
700	→ 14509	14523	14523.00	**14509**	14544.50	873.62
800	→ 17542	17571	17571.00	**17542**	17545.40	449.37
900	→ 20993	21002	21002.00	**20993**	20998.10	316.32
1000	→ 25273	25274	25274.00	**25273**	25273.00	64.70
1087	30326	**30326**	30326.00	**30326**	30326.00	2.67

Table 3: Results for the 4-regular graphs g400-4-01, g400-4-05, g1000-4-01, and g1000-4-05.

(a) 4-regular graph g400-4-01 (400 vertices, 800 edges)

k	**bks**	ACO		HyEA		
		best	**avg**	**best**	**avg**	**avg. time**
2	8	**8**	8.00	**8**	8.00	0.02
40	563	**563**	563.00	**563**	563.70	1.31
80	1304	**1304**	1304.85	**1304**	1305.40	2.29
120	→ 2134	2135	2139.45	**2134**	2134.00	7.70
160	3062	**3062**	3065.95	**3062**	3062.00	4.68
200	4086	**4086**	4086.00	**4086**	4087.70	14.41
240	→ 5224	5225	5228.80	**5224**	5225.30	12.03
280	6487	**6487**	6488.10	**6487**	6487.00	6.94
320	7882	**7882**	7882.00	**7882**	7882.00	4.71
360	9468	**9468**	9468.00	**9468**	9468.00	7.76
398	11433	**11433**	11433.00	**11433**	11433.00	0.10

(b) 4-regular graph g400-4-05 (400 vertices, 800 edges)

k	**bks**	ACO		HyEA		
		best	**avg**	**best**	**avg**	**avg. time**
2	4	**4**	4.00	**4**	4.00	0.02
40	673	**673**	673.00	676	684.60	1.70
80	1445	**1445**	1455.45	1449	1453.80	5.08
120	2293	**2293**	2303.05	**2293**	2294.70	10.87
160	3193	**3193**	3203.70	3195	3196.00	12.54
200	4156	**4156**	4165.75	**4156**	4156.30	14.15
240	→ 5198	5202	5213.30	**5198**	5198.60	19.26
280	6350	**6350**	6361.15	6353	6354.20	22.33
320	7682	**7682**	7682.00	**7682**	7682.00	3.18
360	9249	**9249**	9249.00	**9249**	9249.00	5.66
398	11236	**11236**	11236.00	**11236**	11236.00	0.12

(c) 4-regular graph g1000-4-01 (1000 vertices, 2000 edges)

k	**bks**	ACO		HyEA		
		best	**avg**	**best**	**avg**	**avg. time**
2	6	**6**	6.00	**6**	6.00	0.07
100	1523	**1523**	1564.85	1524	1527.40	17.83
200	→ 3308	3329	3367.10	**3308**	3311.50	47.38
300	→ 5325	5333	5367.30	**5325**	5326.80	119.77
400	7581	**7581**	7595.65	7583	7593.90	204.75
500	10052	**10052**	10066.65	10056	10062.90	304.23
600	12708	**12708**	12725.75	12712	12715.80	382.59
700	15675	**15675**	15675.00	**15675**	15678.10	664.70
800	→ 19023	19037	19037.65	**19023**	19028.40	393.80
900	→ 22827	22830	22830.00	**22827**	22827.00	68.06
998	27946	**27946**	27946.00	**27946**	27946.00	0.99

(d) 4-regular graph g1000-4-05 (1000 vertices, 2000 edges)

k	**bks**	ACO		HyEA		
		best	**avg**	**best**	**avg**	**avg. time**
2	7	**7**	7.00	**7**	7.00	0.05
100	→ 1652	1653	1665.00	**1652**	1653.60	19.67
200	→ 3620	3627	3665.30	**3620**	3623.10	66.00
300	→ 5801	5825	5836.90	**5801**	5807.10	171.47
400	→ 8206	8230	8233.65	**8206**	8212.30	216.24
500	→ 10793	10801	10810.85	**10793**	10795.70	348.94
600	→ 13584	13592	13606.75	**13584**	13587.80	307.26
700	→ 16682	16686	16688.15	**16682**	16686.30	338.49
800	→ 20076	20078	20078.00	**20076**	20077.90	351.13
900	24029	**24029**	24029.00	24033	24037.40	171.95
998	29182	**29182**	29182.00	**29182**	29182.00	1.96

Table 4: Results for random graphs steinc5, steinc15, steind5, and steind15.

(a) Random graph steinc5 (500 vertices, 625 edges)

k	**bks**	ACO		HyEA		
		best	**avg**	**best**	**avg**	**avg. time**
2	5	**5**	5.00	**5**	5.00	0.02
50	→ 772	774	820.15	**772**	773.10	1.24
100	1712	**1712**	1734.65	**1712**	1712.00	3.00
150	2865	**2865**	2888.15	**2865**	2865.00	3.92
200	4273	**4273**	4273.00	4279	4284.40	11.29
250	→ 5945	5952	5955.20	**5945**	5946.70	7.01
300	7938	**7938**	7938.00	**7938**	7938.00	5.18
350	→ 10236	10247	10248.20	**10236**	10238.20	10.44
400	→ 12964	12965	12965.00	**12964**	12967.20	10.64
450	16321	**16321**	16321.00	**16321**	16321.00	8.62
498	20485	**20485**	20485.00	**20485**	20485.00	0.04

(b) Random graph steinc15 (500 vertices, 2500 edges)

k	**bks**	ACO		HyEA		
		best	**avg**	**best**	**avg**	**avg. time**
2	2	**2**	2.00	**2**	2.00	0.01
50	208	**208**	208.00	**208**	208.00	1.53
100	481	**481**	488.45	**481**	481.20	41.46
150	802	**802**	809.70	**802**	802.60	59.94
200	1182	**1182**	1185.80	**1182**	1182.20	111.91
250	→ 1625	1625	1630.15	**1625**	1626.40	203.61
300	2148	**2148**	2148.00	**2148**	2148.00	7.50
350	→ 2795	2795	2796.95	**2795**	2795.00	13.49
400	3571	**3571**	3571.00	**3571**	3571.00	31.68
450	4553	**4553**	4553.00	**4553**	4553.00	3.03
498	5973	**5973**	5973.00	**5973**	5973.00	0.28

(c) Random graph steind5 (1000 vertices, 1250 edges)

k	**bks**	ACO		HyEA		
		best	**avg**	**best**	**avg**	**avg. time**
2	3	**3**	3.00	**3**	3.00	0.06
100	1503	**1503**	1526.35	**1503**	1508.20	7.15
200	→ 3442	3452	3456.50	**3442**	3446.60	39.72
300	→ 5817	5829	5873.45	**5817**	5826.50	75.05
400	→ 8691	8695	8716.15	**8691**	8700.30	83.01
500	→ 12056	12062	12085.70	**12056**	12060.10	100.58
600	→ 15916	15933	15933.00	**15916**	15921.10	146.02
700	→ 20511	20520	20539.45	**20511**	20513.80	129.49
800	26053	**26053**	26053.00	**26053**	26053.00	33.79
900	32963	**32963**	32963.00	**32963**	32963.00	37.68
998	41572	**41572**	41572.00	**41572**	41572.00	2.92

(d) Random graph steind15 (1000 vertices, 5000 edges)

k	**bks**	ACO		HyEA		
		best	**avg**	**best**	**avg**	**avg. time**
2	2	**2**	2.00	**2**	2.00	0.05
100	455	**455**	455.00	**455**	455.00	7.52
200	→ 1018	1029	1038.90	**1018**	1018.70	63.51
300	→ 1674	1680	1680.00	**1674**	1674.50	139.57
400	→ 2446	2451	2458.70	**2446**	2447.70	173.52
500	→ 3365	3366	3369.15	**3365**	3365.20	361.29
600	→ 4420	4423	4424.05	**4420**	4420.00	176.68
700	→ 5685	5686	5686.00	**5685**	5685.00	62.22
800	7236	**7236**	7236.00	**7236**	7236.00	154.22
900	9248	**9248**	9248.00	**9248**	9248.00	10.97
998	12504	**12504**	12504.00	**12504**	12504.00	1.10

Table 5: Results for grid graphs bb45x5_1, bb45x5_2, bb100x10_1, bb100x10_2, bb50x50_1, bb50x50_2, and random graphs steine5, and le450_15a.

(a) Grid graph bb45x5_1 (225 vertices)

k	bks	HyEA		
		best	avg	avg. time
2	2	**2**	2.00	0.01
20	306	**306**	308.40	0.38
40	695	697	707.70	0.85
60	1115	1117	1131.20	0.87
80	→ 1568	**1568**	1586.10	0.94
100	1979	**1979**	2003.70	1.32
120	→ 2450	**2450**	2480.70	1.58
140	3028	3044	3055.10	2.28
160	3702	3711	3722.30	1.89
180	→ 4474	**4474**	4493.40	2.14
200	5461	**5461**	5462.80	0.61
220	6718	**6718**	6718.00	0.40
223	6946	**6946**	6946.00	0.02

(b) Grid graph bb45x5_2 (225 vertices)

k	bks	HyEA		
		best	avg	avg. time
2	8	**8**	8.00	0.01
20	302	**302**	302.00	0.07
40	654	**654**	654.00	0.20
60	1122	**1122**	1122.00	0.33
80	1617	**1617**	1620.90	0.60
100	→ 2129	**2129**	2131.60	0.71
120	→ 2633	**2633**	2646.50	0.89
140	3174	**3174**	3182.90	1.10
160	→ 3757	**3757**	3762.00	1.70
180	4458	**4458**	4458.20	2.13
200	5262	**5262**	5262.50	2.14
220	6347	**6347**	6349.80	1.27
223	6568	**6568**	6568.00	0.02

(c) Grid graph bb100x10_1 (1000 vertices)

k	bks	HyEA		
		best	avg	avg. time
2	3	**3**	3.00	0.08
100	1601	**1601**	1601.00	7.07
200	→ 3520	**3520**	3521.20	25.87
300	→ 5511	**5511**	5542.60	41.39
400	→ 7588	**7588**	7614.80	133.02
500	→ 9961	**9961**	9984.00	190.60
600	→ 12444	**12444**	12463.50	352.62
700	→ 15296	**15296**	15317.10	313.39
800	→ 18670	**18670**	18674.50	241.32
900	→ 22732	**22732**	22738.00	162.07
998	28316	**28316**	28316.00	2.23

(d) Grid graph bb100x10_2 (1000 vertices)

k	bks	HyEA		
		best	avg	avg. time
2	4	**4**	4.00	0.06
100	→ 1661	**1661**	1666.60	5.01
200	→ 3618	**3618**	3635.90	28.57
300	→ 5435	**5435**	5458.10	56.44
400	→ 7531	**7531**	7558.10	131.65
500	→ 9861	**9861**	9876.60	171.25
600	→ 12481	**12481**	12492.00	218.96
700	→ 15599	**15599**	15604.70	387.85
800	→ 19188	**19188**	19198.20	434.18
900	→ 23481	**23481**	23482.00	121.22
998	29474	**29474**	29474.00	3.08

(e) Grid graph bb50x50_1 (2500 vertices)

k	bks	HyEA		
		best	avg	avg. time
2	2	**2**	2.00	0.19
250	→ 3988	**3988**	4018.00	49.59
500	→ 8150	**8150**	8175.20	187.83
750	→ 12551	**12551**	12592.10	449.55
1000	→ 17437	**17437**	17492.00	545.41
1250	→ 22823	**22823**	22855.30	922.59
1500	→ 28683	**28683**	28783.40	832.11
1750	→ 35534	**35534**	35567.80	890.94
2000	→ 43627	**43627**	43645.70	924.21
2250	→ 53426	**53426**	53432.90	750.46
2498	67141	**67141**	67141.00	64.28

(f) Grid graph bb50x50_2 (2500 vertices)

k	bks	HyEA		
		best	avg	avg. time
2	3	**3**	3.00	0.32
250	→ 3612	**3612**	3634.00	47.38
500	→ 7822	**7822**	7846.20	175.00
750	→ 12440	**12440**	12475.90	254.65
1000	→ 17546	**17546**	17614.00	734.63
1250	→ 23448	**23448**	23500.90	879.59
1500	→ 29892	**29892**	29988.60	921.67
1750	→ 37197	**37197**	37252.30	861.37
2000	→ 45673	**45673**	45732.00	872.76
2250	→ 56037	**56037**	56052.10	665.35
2498	70439	**70439**	70439.00	63.98

(g) Random graph steine5 (2500 vertices, 3125 edges))

k	bks	HyEA		
		best	avg	avg. time
2	3	**3**	3.00	0.79
250	→ 3883	**3883**	3893.10	57.24
500	→ 9306	**9306**	9313.80	161.18
750	→ 15818	**15818**	15861.90	213.04
1000	→ 23528	**23528**	23563.00	323.42
1250	→ 32493	**32493**	32524.20	615.53
1500	→ 42769	**42769**	42789.30	533.98
1750	→ 54763	**54763**	54776.50	605.10
2000	→ 68622	**68622**	68628.30	460.59
2250	→ 85366	**85366**	85372.00	148.92
2498	106677	**106677**	106682.10	14.22

(h) Random graph le450_15a (450 vertices, 8168 edges)

k	bks	HyEA		
		best	avg	avg. time
2	2	**2**	2.00	0.01
45	59	**59**	59.20	2.71
90	135	**135**	135.00	1.39
135	226	**226**	226.00	2.11
180	→ 336	**336**	336.00	11.31
225	471	**471**	471.00	12.19
270	630	**630**	630.00	16.66
315	822	**822**	822.00	9.42
360	1060	**1060**	1060.10	7.50
405	1388	**1388**	1388.00	2.10
448	2002	**2002**	2002.00	0.12

Table 6: Results for node weighted graphs.

(a) Grid graphs

instance type	k	**bkr**	VNDS **result**	VNDS **avg. time**	HyACO **result**	HyACO **avg. time**	HyEA **result**	HyEA **avg. time**
30x30	100	8203.50	8571.90	24.00	**8203.50**	65.99	8206.50	63.43
	200	→ 17766.60	17994.40	88.00	17850.10	89.29	**17766.60**	97.99
	300	28770.90	**28770.90**	126.00	28883.90	108.42	28845.40	104.14
	400	42114.00	**42114.00**	80.00	42331.90	133.78	42282.70	117.80
	500	59266.40	**59266.40**	213.00	59541.70	132.52	59551.60	110.31
40x40	150	→ 17461.70	18029.90	112.00	17527.10	211.78	**17461.70**	229.39
	300	→ 37518.50	38965.90	114.00	37623.80	277.70	**37518.50**	297.82
	450	→ 60305.60	61290.10	261.00	60417.00	270.27	**60305.60**	269.80
	600	86422.30	**86422.30**	261.00	86594.70	187.45	86571.10	295.54
	750	117654.00	**117654.00**	303.00	118570.00	217.23	118603.50	260.62
50x50	250	→ 35677.20	37004.00	228.00	35995.20	171.64	**35677.20**	259.78
	500	→ 76963.20	81065.80	322.00	77309.90	286.66	**76963.20**	267.76
	750	→ 125009.00	128200.00	482.00	125415.00	310.13	**125009.00**	284.98
	1000	181983.00	182220.00	575.00	**181983.00**	316.63	182101.50	313.55
	1250	250962.00	**250962.00**	681.00	253059.00	335.35	252683.10	303.51

(b) Random graphs

instance type	k	**bkr**	VNDS **result**	VNDS **avg. time**	HyACO **result**	HyACO **avg. time**	HyEA **result**	HyEA **avg. time**
3000	300	24181.40	**24181.40**	436.00	24345.70	133.73	24364.30	136.47
	600	58719.70	**58719.70**	575.00	59002.40	158.25	58857.60	150.31
	900	106016.40	**106016.40**	177.00	106330.00	164.47	106040.50	116.75
	1200	166948.40	**166948.40**	154.00	167214.00	178.26	166949.40	111.35
	1500	241335.60	**241335.60**	144.00	241569.00	145.25	241338.50	102.68
4000	400	32200.40	**32200.40**	791.00	32589.10	152.83	32828.10	133.42
	800	78755.70	**78755.70**	871.00	79468.10	156.72	79229.20	141.36
	1200	142460.00	**142460.00**	740.00	143259.00	145.11	142578.40	156.61
	1600	224259.70	**224259.70**	316.00	225010.00	171.29	224331.80	141.11
	2000	324681.30	**324681.30**	220.00	325299.00	158.60	324705.70	163.90
5000	500	57725.30	**57725.30**	84.00	58531.10	154.29	59678.40	127.30
	1000	152660.80	**152660.80**	797.00	154857.00	185.83	154060.30	218.43
	1500	293084.80	**293084.80**	789.00	295327.00	241.11	293462.10	238.52
	2000	482370.20	**482370.20**	575.00	484567.00	270.63	482517.40	281.37
	2500	720042.90	720064.10	462.00	721700.00	297.02	720094.10	321.69