Ant System for the k-Cardinality Tree Problem

Thang N. Bui and Gnanasekaran Sundarraj

Department of Computer Science
The Pennsylvania State University at Harrisburg
Middletown, PA 17057
{tbui,gxs241}@psu.edu

Abstract. This paper gives an algorithm for finding the minimum weight tree having k edges in an edge weighted graph. The algorithm combines a search and optimization technique based on pheromone with a weight based greedy local optimization. Experimental results on a large set of problem instances show that this algorithm matches or surpasses other algorithms including an ant colony optimization algorithm, a tabu search algorithm, an evolutionary algorithm and a greedy-based algorithm on all but one of the 138 tested instances.

1 Introduction

Let $G = (V, E)$ be a graph with vertex set V, edge set E and a weight function $w : E \to \mathbb{R}^+$ assigning a weight for each edge of G. A *k-cardinality tree* of G is a subgraph of G that is a tree having exactly k edges. The *weight* of a k-cardinality tree T is the sum of the weights of all the edges in T. The *k-cardinality tree problem* is defined as follows.

Input: Edge-weighted graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}^+$ and an integer k, where $1 \leq k \leq |V| - 1$.
Output: The minimum weight k-cardinality tree of G.

The k-cardinality tree problem was first described by Hamacher, Jornsten and Maffioli in [12] who also proved this problem to be strongly NP-hard. It remains NP-hard even if $w : E \to \{1, 2, 3\}$ [14]. However, it is solvable in polynomial time if the range of w has cardinality 2 [14]. This problem arises in various areas such as facility layout [9], graph partitioning [10], quorum cast routing [5], telecommunications [11], and matrix decomposition [4].

Several heuristic and meta-heuristic algorithms have been developed for this problem. Under the heuristic category, an integer programming approach is given in [8], and a Branch and Bound approach is given in [5]. These heuristic algorithms are based on the greedy and dual greedy strategies in addition to dynamic programming technique. Recent meta-heuristic approaches for this problem include ant colony optimization [2], evolutionary computation [3], tabu search [3], and variable neighborhood search [13].

In this paper, we present an ant system algorithm for the k-cardinality tree problem. We test the algorithm on a number of benchmark graphs and for a

K. Deb et al. (Eds.): GECCO 2004, LNCS 3102, pp. 36–47, 2004.

large number of k values. We compare our results against a number of existing heuristics including an ant colony optimization algorithm, an evolutionary algorithm, a tabu search algorithm and a greedy-based algorithm. The experimental results show that our algorithm performs very well against these algorithms, matching or surpassing all of them in all but one of the 138 problem instances. In fact, for more than half of the instances, our algorithm found solutions that are better than the previously best known solutions.

The rest of the paper is organized as follows. Section 2 describes the algorithm in detail. The experimental results comparing our algorithm against other known algorithms are given in Section 3 and the conclusion is given in Section 4.

2 Algorithm

The main idea for our algorithm is that a given edge in the input graph by itself may not have a low enough weight to be considered for the solution tree, but when combined with other neighboring edges without creating a cycle, this set of connected edges may have the lowest total weight compared to other equicardinality, acyclic, connected set of edges in its neighborhood. We use ants to discover such connected acyclic sets of edges in the graph that can be combined to obtain an optimal k-cardinality tree. In fact, it has been shown that the connectivity requirement is the crux of the difficulty in this problem [7].

The algorithm, called ASkCT and given in Figure 1, consists of two main phases. The first phase has two stages: discovery and construction. In the discovery stage ants are used to discover a potential set of edges from which a small weight k-cardinality tree can be constructed. In the construction stage a greedy algorithm similar to the Kruskal algorithm for finding the minimum cost spanning tree is used to construct a k-cardinality tree of small weight from the set of potential edges produced by the discovery stage. The greedy strategy used in this construction stage is based on the pheromone left on the edges by the ants in the discovery stage.

The second phase of the algorithm consists of a sequence of local optimization stages designed to reduce the weight of the k-cardinality tree produced by the first phase. In what follows, we describe each of the two phases in detail.

2.1 The Discovery Stage

In this stage, we let each ant, starting from a given vertex, to discover a path consisting of more than one edge. We then let the ant move to the other end vertex of the path while allowing the ant to deposit pheromone along the path. We call such a move a *step* and number of edges in that move the *step size*. The amount of pheromone deposited on each edge in the path is inversely proportional to the total weight of the edges in the path. The edges in these paths are stored in the ant's 'memory' so that they will not be rediscovered by the same ant when it is trying to find the next best path from the other end vertex of

```
ASkCT(G = (V, E), w)   // w is the weight function
  Phase 1.
  for s = minStepSize to maxStepSize   // s is the step size
      CandidateEdges ← Discover(G, w, s)          // Discovery Stage
      T ← ConstructTree(CandidateEdges, G, w)     // Construction Stage
      if T is better than the current best tree T* // Remember the best
         T* ← T
  end-step s

  Phase 2.
  T* ← Optimize(T*)
  return T*
```

Fig. 1. The ASkCT algorithm

```
Discover(G = (V, E), w, s)) //w is the weight function, s is the step size
  InitializePheromone()
  for i = 1 to 50
     distribute ants on the vertices
     for each ant a
        for step = 1 to numSteps
           find the best path of length s from the current vertex
                without using the edges in a's memory
           apply pheromone on the s edges of this path
           add these edges to a's memory
           move a to the other end of the path
        end-step
     end-ant
     apply pheromone evaporation
     clear ants' memory
  end-for
```

Fig. 2. The Discover Algorithm

the already discovered path and also to make sure that no cycle is created. This stage is accomplished by the Discover algorithm given in Figure 2.

The Discover algorithm starts by applying an initial amount of pheromone on all the edges and distributing the ants randomly on the vertices. For our experiment this initial amount of pheromone is set to 0.5, the pheromone evaporation rate is set to 0.01, and the number of ants is set to 20% of the number of vertices in graph. The algorithm then runs through a number of iterations, which is set to 50 for our experiment. At the end of each iteration, the memory of each ant is cleared and each ant starts the next iteration from a randomly chosen vertex. In each iteration, an ant takes `numSteps` steps. We set `numSteps` to 2 in our current implementation.

2.2 The Tree Construction Stage

We use a modified version of the Kruskal algorithm [6] for constructing the tree. A k-cardinality tree is extracted from the candidate edges based on the pheromone left by the ants in the discovery stage.

First, the edges are sorted into order of decreasing pheromone values. The algorithm maintains a collection of disjoint sets as in the normal implementation of the Kruskal algorithm, each disjoint set is a tree. Starting from the edge that has the highest pheromone value, edges are added to the disjoint sets that contain one of their end vertices. For any edge, if there is no disjoint set that has one of its end vertices, a new disjoint set is created and the edge is added to that set. If the end vertices of an edge are in two different disjoint sets, the sets are merged into a single set and the edge is added to the merged set. If a disjoint set has both the end vertices of an edge, the algorithm checks the loop that is formed by the addition of this new edge. If the new edge is better in weight than any other edge in the loop, that edge is replaced by the new edge. Once an edge is added to a disjoint set, the sets are checked to see if any of them contain k or more edges. If there is one, a k-cardinality tree is constructed out of these edges. If the size is more than k, the leaf edges are trimmed off in a greedy manner until there are only k edges left, i.e., higher weight leaf edges are removed first. A *leaf edge* is an edge one of whose end point is a leaf. The newly constructed tree is compared with the current best tree, and the smaller of the two is kept as the current best tree. Once the tree is constructed from a disjoint set, the set is marked as processed. New edges will be added to the set that is marked as processed, but no new tree will be constructed from that disjoint set until it is merged with another disjoint set. The tree obtained at the end is considered as the best for that step-size. The algorithm is given in Figure 3.

```
ConstructTree(CandidateEdges, G = (V, E), w)
   Sort the edges in CandidateEdges into decreasing pheromone values
   for each edge e in the sorted order
      Find the disjoint sets that contain the end vertices of e
      if there is none
         create a new set, add e to it, mark the set unprocessed
      if there is only one set containing one of the end vertices of e
         add e to it
      if there is only one set containing both end vertices
         try to replace an edge in the loop formed; otherwise discard e
      if there are two different sets, each containing one end vertex of e
         merge the two sets
         add e to the merged set
         mark the merged set unprocessed

      for each disjoint set A
         if A has k or more edges
            extract a k-cardinality tree by trimming greedily if needed
            if the new tree has a lower weight than the best known so far
               discard the previous best and take the new tree as the best
            mark A processed
      end-disjoint-set
   end-edge
   return the best k-cardinality tree
```

Fig. 3. The ConstructTree Algorithm

It can be noted from the ASkCT algorithm given in Figure 1 that the discovery and the tree construction stages are repeated for various step sizes, that is, the step size is varied from `minStepSize` to `maxStepSize`. For our experiment `minStepSize` is set to 2 and `maxStepSize` is set to 3. The algorithm then picks the best tree out of these step sizes to pass on to the next phase to be optimized.

2.3 The Local Optimization Phase

In the local optimization phase, we apply a sequence of greedy algorithms to reduce the weight of the k-cardinality tree produced by the previous phase. Unlike the previous phase, the strategy used here is based on the weight of the edges not the pheromone. The main idea here is to swap edges in and out of the tree so that the tree weight is reduced. Specifically, this phase consists of four stages: (i) the tree is grown first and then shrunk back, in a greedy manner, to its original size, (ii) the tree is shrunk and then grown back, in a greedy manner, to its original size, (iii) the tree is shrunk and then grown back, in a depth first manner, to its original size, and (iv) the tree is split into two by removing the highest weight edge, and the resulting trees are grown back until they meet or one of them has k edges. The algorithm is given in Figure 4.

In these stages, when a tree is grown by i edges in a greedy manner, we select i smallest weight edges among the edges that are connected to the tree by one endpoint only, and add them to the tree. Similarly, when a tree is trimmed by i leaf edges in a greedy manner, we select the i highest weight edges among the leaf edges of the tree and remove them. When a tree is grown by i edges in a depth-first manner, we select by using a depth-first search the lowest weight path of length i that has one end connected to the tree and add the edges in the path to the tree.

These stages were chosen so that the effect of the individual stages are complementary to each other. In Stages 1–3, we try to replace the leaf edges, whereas in Stage 4 we try to replace the non-leaf edges, beginning at the highest weight edge. In order to achieve a balance between the quality of the results and the overall running time of the algorithm, only 30% of the edges in the k-cardinality tree were considered for replacement in each of these stages except Stage 3.

We observed that the k-cardinality tree problem is usually more difficult when the cardinality k is small but not too small. The local optimization algorithm accounts for this in the third for loop with a variable number of iterations based on the size of k. Specifically, the function $\alpha(k)$ in the third for loop in Figure 4 is defined as follows.

$$\alpha(k) = \begin{cases} 6, & \text{if } 0.05|V| < k < 0.35|V|, \\ 3, & \text{otherwise.} \end{cases}$$

where V is the vertex set of the input graph.

```
LocalOptimize(T)    // T is a k-cardinality tree
Stage 1.
   T* ← T
   for i = 1 to ⌊3k/10⌋
      grow T by adding i edges in a greedy manner
      trim T by removing i leaf edges in a greedy manner
      if w(T) < w(T*)
         T* ← T
   end-for

Stage 2.
   T ← T*
   for i = 1 to ⌊3k/10⌋
      trim T by removing i leaf edges in a greedy manner
      grow T by adding i edges in a greedy manner
      if w(T) < w(T*)
         T* ← T
   end-for

Stage 3.
   T ← T*
   for i = α(k) downto 2
      trim T by removing i leaf edges in a greedy manner
      find the lowest weight path of length i with one end connected to T
      add the edges of this path to T
      if w(T) < w(T*)
         T* ← T
   end-for

Stage 4.
   T ← T*
   sort the edges of T into decreasing order of weight
   for i = 1 to ⌊3k/10⌋
      remove the ith highest weight edge from T to obtain two
         trees T1 and T2
      grow T1 and T2 independently in a greedy manner
          until they meet or one of them has k edges
      let T' be the resulting tree
      if T' has more than k edges,
         trim off the excess leaf edges in a greedy manner
      if w(T') < w(T*)
         T* ← T'
   end-for
   return T*
```

Fig. 4. The LocalOptimize algorithm

3 Experimental Results

In this section, we describe the results of running our algorithm on a collection of benchmark graphs for this problem, and compare them against the current best known results from four other algorithms: an ACO algorithm [2], an evolutionary algorithm [3], a tabu search algorithm [1], and a greedy based algorithm [7].

Our algorithm was implemented in C++ and run on a PC with Pentium IV 2.4GHz processor and 512MB of RAM. We tested our algorithm on three different classes of graphs: random graphs, grid graphs and Steiner graphs. There are four graphs in each class, for a total of twelve graphs. In order to be able to compare the performance of our algorithm against others, we selected these

graphs from KCTLIB, a library for the edge-weighted k-cardinality tree problem, maintained by C. Blum and M. Blesa at http://iridia.ulb.ac.be/~cblum/kctlib/. We also used the same set of values for k in each graph and the same number of runs, which is 20, for each of these k values as used by the authors of KCTLIB. Since the configuration of the system used to run the other algorithms was not available, we could not compare the running time of our algorithm against others.

Of the 138 instances that we tested, our algorithm ASkCT matches the previous best known results in 61 cases. It provides better results than previously known in 76 cases. ASkCT did not match the best known result for the one remaining case. The difference in this case has an absolute difference of 7 or 0.46% of the best known value. The results shown in Tables 1 through 6 list the best (w_{best}), the average (w_{avg}), the standard deviation (σ), and the average running time in seconds (t_{avg}) for our algorithm. These tables also include the previous best-known values for each of the k values. It can be observed from these tables that the standard deviations are very small for the most part. In fact, the standard deviations for these results are no more than 3.5% of the best known value. The results shown in Tables 7 through 12 compare our best values against the best values from other algorithms. It should be noted that the previously best known results were not achieved by any one single algorithm alone. They were the bests obtained among the four algorithms. The data for these four algorithms were obtained from KCTLIB.

From our experiments, we observed that the influence of the local optimization phase on the final results were minimal. It improved the results by no more than 5-10%. For many k values, we were able to obtain the same results without the local optimization.

In our implementation, the values for `minStepSize` and `maxStepSize` in the ASkCT algorithm and the value for `numSteps` in the Discover algorithm were chosen based on our experiments on two graphs. The same is true of the values for $\alpha(k)$ in the LocalOptimize algorithm.

Table 1. ASkCT solution quality on random graphs with 400 vertices

g400-4-01 (400 vertices, 800 edges)						g400-4-05 (400 vertices, 800 edges)					
k	Best known	w_{best}	w_{avg}	σ	t_{avg} (sec)	k	Best known	w_{best}	w_{avg}	σ	t_{avg} (sec)
2	8	8	8.00	0.00	0.02	2	4	4	4.00	0.00	0.02
40	563	563	563.00	0.00	3.99	40	675	673	673.00	0.00	3.95
80	1304	1304	1304.85	0.36	6.35	80	1457	1445	1455.45	6.50	6.30
120	2137	2135	2139.45	1.40	11.00	120	2295	2293	2303.05	5.75	11.88
160	3066	3062	3065.95	1.91	5.18	160	3197	3193	3203.70	5.75	5.21
200	4105	4086	4086.00	0.00	7.12	200	4169	4156	4165.75	3.83	7.88
240	5238	5225	5228.80	0.87	15.87	240	5209	5202	5213.30	4.15	15.10
280	6499	6487	6488.10	0.83	24.63	280	6372	6350	6361.15	0.00	21.83
320	7888	7882	7882.00	0.00	22.52	320	7682	7682	7682.00	0.00	29.52
360	9471	9468	9468.00	0.00	38.90	360	9250	9249	9249.00	0.00	38.82
398	11433	11433	11433.00	0.00	35.20	398	11236	11236	11236.00	0.00	35.66

Table 2. ASkCT solution quality on random graphs with 1000 vertices

g1000-4-01 (1000 vertices, 2000 edges)						g1000-4-05 (1000 vertices, 2000 edges)					
k	Best known	w_{best}	w_{avg}	σ	t_{avg} (sec)	k	Best known	w_{best}	w_{avg}	σ	t_{avg} (sec)
2	6	6	6.00	0.00	0.07	2	7	7	7.00	0.00	0.07
100	1528	1523	1564.85	22.86	8.51	100	1654	1653	1665.00	5.92	5.06
200	3341	3329	3367.10	10.49	59.51	200	3639	3627	3665.30	10.18	26.13
300	5334	5333	5367.30	17.38	109.96	300	5842	5825	5836.90	5.38	122.01
400	7609	7581	7595.65	4.13	75.94	400	8302	8230	8233.65	1.06	79.76
500	10104	10052	10066.65	6.89	157.73	500	10893	10801	10810.85	5.30	187.49
600	12794	12708	12725.75	5.51	316.03	600	13725	13592	13606.75	4.80	305.38
700	15767	15675	15675.00	0.00	581.87	700	16803	16686	16688.15	0.78	569.16
800	19079	19037	19037.65	0.48	685.38	800	20128	20078	20078.00	0.00	756.91
900	22838	22830	22830.00	0.00	840.25	900	24035	24029	24029.00	0.00	866.67
998	27946	27946	27946.00	0.00	825.71	998	29182	29182	29182.00	0.00	834.94

Table 3. ASkCT solution quality on grid graphs with 225 vertices

bb15x15_1 (225 vertices, 420 edges)						bb15x15_2 (225 vertices, 420 edges)					
k	Best known	w_{best}	w_{avg}	σ	t_{avg} (sec)	k	Best known	w_{best}	w_{avg}	σ	t_{avg} (sec)
2	2	2	2.00	0.00	0.01	2	6	6	6.00	0.00	0.01
20	257	257	258.00	3.00	0.17	20	253	253	253.00	0.00	0.16
40	642	642	644.40	1.20	0.39	40	585	585	624.10	20.54	0.32
60	977	977	1005.50	9.38	0.75	60	927	927	986.05	30.61	0.61
80	1335	1335	1429.15	29.12	0.95	80	1290	1290	1348.35	19.19	1.26
100	1762	1762	1780.05	15.97	1.63	100	1686	1686	1726.25	11.88	0.90
120	2235	2235	2262.80	9.40	3.89	120	2120	2120	2143.55	8.46	2.81
140	2781	2781	2798.10	6.46	4.95	140	2634	2634	2639.60	4.07	3.76
160	3417	3417	3423.00	7.03	3.37	160	3260	3250	3272.75	9.59	3.33
180	4158	4158	4162.15	3.71	5.88	180	3915	3915	3915.00	0.00	4.94
200	5040	5040	5040.95	0.22	4.17	200	4718	4718	4718.00	0.00	6.15
220	6176	6176	6176.00	0.00	7.37	220	5862	5862	5862.00	0.00	7.17
223	6400	6400	6400.00	0.00	7.30	223	6101	6101	6101.00	0.00	7.25

Table 4. ASkCT solution quality on grid graphs with 1089 vertices

bb33x33_1 (1089 vertices, 2112 edges)						bb33x33_2 (1089 vertices, 2112 edges)					
k	Best known	w_{best}	w_{avg}	σ	t_{avg} (sec)	k	Best known	w_{best}	w_{avg}	σ	t_{avg} (sec)
2	3	3	3.00	0.00	0.08	2	3	3	3.00	0.00	0.08
100	1587	1587	1594.60	1.74	12.48	100	1524	1531	1568.65	15.43	6.27
200	3386	3366	3466.35	35.55	48.79	200	3378	3316	3530.60	94.65	35.63
300	5235	5235	5320.45	31.39	165.52	300	5289	5275	5360.10	28.81	116.51
400	7192	7166	7224.80	14.77	103.83	400	7366	7340	7582.05	98.69	75.56
500	9461	9256	9327.60	29.78	200.01	500	9626	9514	9624.70	55.98	173.90
600	11743	11579	11579.00	0.00	319.43	600	12113	11879	11889.30	2.45	319.74
700	14556	14309	14313.35	0.78	518.59	700	14664	14523	14523.00	0.00	552.12
800	17606	17399	17399.00	0.00	716.44	800	17667	17571	17571.00	0.00	723.64
900	21057	20921	20921.00	0.00	776.63	900	21037	21002	21002.00	0.00	774.41
1000	25235	25199	25199.00	0.00	1103.54	1000	25275	25274	25274.00	0.00	1145.04
1087	30417	30417	30417.00	0.00	1026.62	1087	30326	30326	30326.00	0.00	1044.47

4 Conclusion

In this paper, we gave an efficient ant system algorithm, called ASkCT, for the k-cardinality tree problem. The algorithm combines a search and optimization

Table 5. ASkCT solution quality on Steiner graphs with 500 vertices

steinc5 (500 vertices, 625 edges)						steinc15 (500 vertices, 2500 edges)					
k	Best known	w_{best}	w_{avg}	σ	t_{avg} (sec)	k	Best known	w_{best}	w_{avg}	σ	t_{avg} (sec)
2	5	5	5.00	0.00	0.02	2	2	2	2.00	0.00	0.05
50	774	774	820.15	11.85	2.09	50	208	208	208.00	0.00	15.02
100	1712	1712	1734.65	6.92	5.44	100	481	481	488.45	1.77	27.86
150	2871	2865	2888.15	5.92	14.94	150	802	802	809.70	3.15	28.90
200	4279	4273	4273.00	0.00	8.29	200	1183	1182	1185.80	0.87	17.75
250	5965	5952	5955.20	4.26	15.67	250	1628	1628	1630.15	1.06	24.01
300	7986	7938	7938.00	0.00	24.67	300	2150	2148	2148.00	0.00	46.89
350	10292	10247	10248.20	0.98	39.97	350	2798	2796	2796.95	0.22	44.00
400	12992	12965	12965.00	0.00	59.48	400	3571	3571	3571.00	0.00	64.86
450	16321	16321	16321.00	0.00	32.62	450	4553	4553	4553.00	0.00	134.63
498	20485	20485	20485.00	0.00	70.20	498	5973	5973	5973.00	0.00	142.62

Table 6. ASkCT solution quality on Steiner graphs with 1000 vertices

steind5 (1000 vertices, 1250 edges)						steind15 (1000 vertices, 5000 edges)					
k	Best known	w_{best}	w_{avg}	σ	t_{avg} (sec)	k	Best known	w_{best}	w_{avg}	σ	t_{avg} (sec)
2	3	3	3.00	0.00	0.04	2	2	2	2.00	0.00	0.16
100	1515	1503	1526.35	7.79	11.16	100	455	455	455.00	0.00	80.15
200	3469	3452	3456.50	1.50	51.92	200	1035	1029	1038.90	3.82	129.54
300	5897	5829	5873.45	16.03	139.64	300	1691	1680	1680.00	0.00	265.45
400	8886	8695	8716.15	7.48	85.97	400	2472	2451	2458.70	4.12	165.50
500	12172	12062	12085.70	13.09	184.10	500	3382	3366	3369.15	1.11	343.72
600	16091	15933	15933.00	0.00	281.90	600	4434	4423	4424.05	0.97	528.90
700	20646	20520	20539.45	10.20	449.47	700	5704	5686	5686.00	0.00	994.40
800	26103	26053	26053.00	0.00	769.75	800	7241	7236	7236.00	0.00	1530.38
900	32963	32963	32963.00	0.00	599.64	900	9256	9248	9248.00	0.00	1726.35
998	41572	41572	41572.00	0.00	631.28	998	12504	12504	12504.00	0.00	1658.98

Table 7. Results for all algorithms on random graphs with 400 vertices.

g400-4-01 (400 vertices, 800 edges)							g400-4-05 (400 vertices, 800 edges)						
k	Best known	ASkCT Best	ACO Best	EC Best	TS Best	KCP Best	k	Best known	ASkCT Best	ACO Best	EC Best	TS Best	KCP Best
2	8	**8**	**8**	**8**	**8**	**8**	2	4	**4**	**4**	**4**	**4**	**4**
40	563	**563**	**563**	**563**	**563**	592	40	675	**673**	676	676	675	739
80	1304	**1304**	**1304**	1305	1307	1392	80	1457	**1445**	1457	1460	1466	1601
120	2137	**2135**	2137	2140	2140	2285	120	2295	**2293**	2295	2314	2318	2451
160	3066	**3062**	3066	3071	3070	3198	160	3197	**3193**	3197	3217	3217	3389
200	4105	**4086**	4105	4117	4112	4249	200	4169	**4156**	4169	4171	4171	4400
240	5238	**5225**	5247	5255	5238	5410	240	5209	**5202**	5209	5217	5216	5525
280	6499	**6487**	6509	6514	6499	6666	280	6372	**6350**	6383	6372	6378	6533
320	7888	**7882**	7903	7892	7888	8048	320	7682	**7682**	7713	**7682**	**7682**	7869
360	9471	**9468**	9494	9472	9471	9553	360	9250	**9249**	9295	9256	9250	9262
398	11433	**11433**	11454	**11433**	**11433**	**11433**	398	11236	**11236**	11278	**11236**	**11236**	**11236**

based on pheromone with a weight based optimization. Extensive experimental results show that ASkCT outperforms existing heuristics from different methodologies. We note that the weight based local optimization algorithm can also be used with other algorithms as an extra optimization. Possible future work includes improving the quality of ASkCT even further, particularly, for the cases when k is small compared to the number of vertices in the graph and for

Table 8. Results for all algorithms on random graphs with 1000 vertices

g1000-4-01 (1000 vertices, 2000 edges)							g1000-4-05 (1000 vertices, 2000 edges)						
k	Best known	ASkCT Best	ACO Best	EC Best	TS Best	KCP Best	k	Best known	ASkCT Best	ACO Best	EC Best	TS Best	KCP Best
2	6	**6**	**6**	**6**	**6**	**6**	2	7	**7**	**7**	**7**	**7**	**7**
100	1528	**1523**	1528	1558	1567	1684	100	1654	**1653**	1654	1657	1662	1782
200	3341	**3329**	3341	3445	3438	3652	200	3639	**3627**	3639	3680	3692	3994
300	5334	**5333**	5334	5500	5482	5828	300	5842	**5825**	5842	5875	5923	6320
400	7609	**7581**	7609	7749	7669	8329	400	8302	**8230**	8302	8320	8344	8875
500	10104	**10052**	10114	10104	10125	10837	500	10893	**10801**	10922	10893	10956	11492
600	12794	**12708**	12864	12794	12797	13584	600	13725	**13592**	13780	13743	13725	14392
700	15767	**15675**	15806	15772	15767	16223	700	16803	**16686**	16924	16803	16805	17399
800	19079	**19037**	19232	19090	19079	19494	800	20128	**20078**	20262	20134	20128	20576
900	22838	**22830**	23022	22839	22838	23076	900	24035	**24029**	24226	24045	24035	24272
998	27946	**27946**	28119	**27946**	**27946**	**27946**	998	29182	**29182**	29342	**29182**	**29182**	**29182**

Table 9. Results for all algorithms on grid graphs with 225 vertices

bb15x15_1 (225 vertices, 420 edges)							bb15x15_2 (225 vertices, 420 edges)						
k	Best known	ASkCT Best	ACO Best	EC Best	TS Best	KCP Best	k	Best known	ASkCT Best	ACO Best	EC Best	TS Best	KCP Best
2	2	**2**	**2**	**2**	**2**	**2**	2	6	**6**	**6**	**6**	**6**	**6**
20	257	**257**	**257**	**257**	**257**	267	20	253	**253**	**253**	**253**	**253**	**253**
40	642	**642**	**642**	**642**	**642**	650	40	585	**585**	**585**	**585**	592	620
60	977	**977**	**977**	988	**977**	1154	60	927	**927**	**927**	929	930	1075
80	1335	**1335**	**1335**	1359	1355	1518	80	1290	**1290**	**1290**	1315	1324	1471
100	1762	**1762**	**1762**	1764	1764	1998	100	1686	**1686**	**1686**	1725	1741	1907
120	2235	**2235**	**2235**	**2235**	**2235**	2554	120	2120	**2120**	**2120**	2127	2155	2342
140	2781	**2781**	2783	**2781**	2783	2956	140	2634	**2634**	**2634**	2638	2642	2773
160	3417	**3417**	**3417**	**3417**	3435	3475	160	3260	**3250**	3260	3278	3268	3289
180	4158	**4158**	**4158**	**4158**	4167	4216	180	3915	**3915**	3922	3922	**3915**	3968
200	5040	**5040**	5059	**5040**	5041	5104	200	4718	**4718**	4722	**4718**	**4718**	4867
220	6176	**6176**	6183	**6176**	**6176**	**6176**	220	5862	**5862**	5864	**5862**	**5862**	**5862**
223	6400	**6400**	6401	**6400**	**6400**	**6400**	223	6101	**6101**	6105	**6101**	**6101**	**6101**

Table 10. Results for all algorithms on grid graphs with 1089 vertices

bb33x33_1 (1089 vertices, 2112 edges)							bb33x33_2 (1089 vertices, 2112 edges)						
k	Best known	ASkCT Best	ACO Best	EC Best	TS Best	KCP Best	k	Best known	ASkCT Best	ACO Best	EC Best	TS Best	KCP Best
2	3	**3**	**3**	**3**	**3**	**3**	2	3	**3**	**3**	**3**	**3**	**3**
100	1587	**1587**	**1587**	1595	1615	1802	100	1524	1531	**1524**	1558	1569	1760
200	3386	**3366**	3386	3410	3537	3927	200	3378	**3316**	3378	3475	3431	3761
300	5235	**5235**	**5235**	5261	5433	6038	300	5289	**5275**	5289	5486	5510	5876
400	7192	**7166**	7192	7328	7415	8232	400	7366	**7340**	7366	7497	7562	8571
500	9461	**9256**	9461	9556	9584	11038	500	9626	**9514**	9626	9687	9964	10689
600	11743	**11579**	11743	11946	12056	13263	600	12113	**11879**	12113	12256	12260	13164
700	14556	**14309**	14556	14746	14775	15724	700	14664	**14523**	14664	14884	14979	15808
800	17606	**17399**	17636	17606	17618	18683	800	17667	**17571**	17780	17667	17798	18831
900	21057	**20921**	21266	21057	21072	21862	900	21037	**21002**	21252	21037	21055	22144
1000	25235	**25199**	25582	25235	25264	25882	1000	25275	**25274**	25620	25275	25299	25914
1087	30417	**30417**	30571	**30417**	**30417**	**30417**	1087	30326	**30326**	30467	**30326**	**30326**	**30326**

grid graphs. The grid graph `bb33x33_2` with $k = 100$ is the only instance that ASkCT did not match or surpass the best known value among the tested graphs.

Table 11. Results for all algorithms on Steiner graphs with 500 vertices

steinc5 (500 vertices, 625 edges)							steinc15 (500 vertices, 2500 edges)						
k	Best known	ASkCT Best	ACO Best	EC Best	TS Best	KCP Best	k	Best known	ASkCT Best	ACO Best	EC Best	TS Best	KCP Best
2	5	**5**	**5**	**5**	**5**	**5**	2	2	**2**	**2**	**2**	**2**	**2**
50	774	**774**	**774**	777	783	877	50	208	**208**	**208**	**208**	**208**	229
100	1712	**1712**	**1712**	1714	**1712**	1963	100	481	**481**	482	483	**481**	526
150	2871	**2865**	2871	2896	2892	3163	150	802	**802**	**802**	807	815	923
200	4279	**4273**	4279	4335	4318	4552	200	1183	**1182**	1183	1185	1190	1276
250	5965	**5952**	5983	5992	5965	6221	250	1628	**1628**	**1628**	1633	1633	1731
300	7986	**7938**	8014	8014	7986	8378	300	2150	**2148**	2150	2158	2153	2227
350	10292	**10247**	10315	10371	10292	10711	350	2798	**2796**	2802	2798	2799	2950
400	12992	**12965**	13060	13023	12992	13475	400	3571	**3571**	3578	**3571**	**3571**	3666
450	16321	**16321**	16359	16334	**16321**	16666	450	4553	**4553**	4575	**4553**	4555	4584
498	20485	**20485**	20495	**20485**	**20485**	**20485**	498	5973	**5973**	5986	**5973**	**5973**	**5973**

Table 12. Results for all algorithms on Steiner graphs with 1000 vertices

steind5 (1000 vertices, 1250 edges)							steind15 (1000 vertices, 5000 edges)						
k	Best known	ASkCT Best	ACO Best	EC Best	TS Best	KCP Best	k	Best known	ASkCT Best	ACO Best	EC Best	TS Best	KCP Best
2	3	**3**	**3**	**3**	**3**	**3**	2	2	**2**	**2**	**2**	**2**	**2**
100	1515	**1503**	1515	1570	1587	1790	100	455	**455**	**455**	457	457	491
200	3469	**3452**	3469	3539	3596	4135	200	1035	**1029**	1037	1035	1041	1138
300	5897	**5829**	5897	6031	6027	6766	300	1691	**1680**	1696	1691	1708	1857
400	8886	**8695**	8886	8972	8941	9881	400	2472	**2451**	2552	2487	2472	2684
500	12172	**12062**	12267	12216	12172	13830	500	3382	**3366**	3599	3382	3396	3607
600	16091	**15933**	16091	16125	16139	17190	600	4434	**4423**	4797	4434	4437	4670
700	20646	**20520**	20700	20671	20646	21520	700	5704	**5686**	6034	5704	5707	5889
800	26103	**26053**	26227	26103	26147	27173	800	7241	**7236**	7837	7241	7245	7407
900	32963	**32963**	33119	33006	**32963**	33579	900	9256	**9248**	9771	9256	9276	9388
998	41572	**41572**	41637	**41572**	**41572**	**41572**	998	12504	**12504**	12759	**12504**	**12504**	**12504**

Acknowledgements. The authors would like to thank Christian Blum for helpful discussions and the anonymous reviewers for their valuable comments.

References

1. Blesa, M. J. and F. Xhafa, "A C++ Implementation of Tabu Search for k-Cardinality Tree Problem Based on Generic Programming and Component Reuse," Net.ObjectDsays 2000 Tagungsband, NetObjectDays Forum, Germany, 2000, pp. 648–652.
2. Blum, C., "Ant Colony Optimization for the Edge-Weighted k-Cardinality Tree Problem," Proceedings of the Genetic and Evolutionary Computation Conference, 2002, pp. 27–34.
3. Blum, C. and M. Ehrgott, "Local Search Algorithms for the k-cardinality Tree Problem," Technical report TR/IRIDIA/2001-12, IRIDIA, Université Libre de Bruxelles, Belgium.
4. Borndörfer, R., C. Ferreira and A. Martin, " Decomposing Matrices Into Blocks," SIAM Journal on Optimization, 9(1), 1998, pp. 236–269.
5. Cheung, S. Y. and A. Kumar, "Efficient Quorumcast Routing Algorithms," in Proceedings of INFOCOM'94, Los Alamitos, 1994.

6. Cormen, T. H., C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms,* Second Edition, McGraw-Hill, 2001.
7. Ehrgott, M., 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), 1997, pp. 87–114.
8. Fischetti, M., W. Hamacher, K. Jornsten and F. Maffioli, "Weighted k-Cardinality Trees: Complexity and Polyhedral Structure," Networks, 24, 1994, pp. 11–21.
9. Foulds, L. R. and H. W. Hamacher, "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.
10. Foulds, L. H. Hamacher and J. Wilson, "Integer Programming Approaches to Facilities Layout Models with Forbidden Areas," Annals of Operations Research, 81, 1998, pp. 405–417.
11. Garg, N. and D. Hochbaum, "An $O(\log k)$ Approximation Algorithm for the k Minimum Spanning Tree Problem in the Plane," Algorithmica, 18, 1997, pp. 111–121.
12. Hamacher, H. W., K. Jornsten and F. Maffioli, " Weighted K-Cardinality Trees," Technical Report 91.023, Dept. di Elettronica, Politecnico di Milano, 1991.
13. Mladenovic, N., "Variable Neighborhood Search for the k-Cardinality Tree Problem," Proc. of the Metaheuristics International Conference, MIC'2001, 2001.
14. Ravi, R., R. Sundaram, M. V. Marathe, D. J. Rosenkrantz and S. S. Ravi, "Spanning Trees – Short or Small," SIAM J. on Discrete Mathematics, 9(2), 1996, pp. 178–200.