Revisiting Esau-Williams' Algorithm: On the Design of Local Access Networks[1]

Raja Jothi and Balaji Raghavachari
Department of Computer Science
University of Texas at Dallas
Richardson, Texas 75083-0688
E-mail: {raja,rbk}@utdallas.edu

Corresponding author: Raja Jothi
Department of Computer Science
Eric Jonsson School of Engineering and Computer Science
University of Texas at Dallas
P.O. Box 830688, MS EC31
Richardson, TX 75083-0688
Phone: (972) 918-9387
Fax: (972) 883-2349
Email: raja@utdallas.edu

[1] A preliminary version of this paper appeared as an abstract in the proceedings of the 7th INFORMS Telecommunications Conference, 2004. Research supported in part by the National Science Foundation under grant CCR-9820902.

Abstract

Given a set of nodes, each associated with a positive number denoting the traffic to be routed to a central node (root), the capacitated minimum spanning tree (CMST) problem asks for a minimum cost spanning tree, spanning all nodes, such that the amount of traffic routed from a subtree, linked to the root by an edge, does not exceed the given capacity constraint k. The cost of a tree is defined to be the sum of cost of the edges in the tree. The CMST problem is NP-hard and has been extensively studied for the past 40 years. A major problem with most of the proposed heuristics is that their worst-case running-times may not necessarily be polynomial, i.e., they could be exponential. The most popular and efficient algorithm for the CMST problem is due to Esau and Williams (EW), presented in 1966, with running time $O(n^2 \log n)$. Almost all of the heuristics that have been proposed so far, use the EW algorithm as a benchmark to compare their results. Any other heuristic that outperforms the EW algorithm do so with an enormous increase in running time. In this paper, we propose a modification to the EW algorithm. Our proposed modification guarantees better quality solutions when compared to that produced by the EW algorithm while still maintaining the worst-case run-time complexity at $O(n^2 \log n)$. Experimental results on benchmark instances show that the modified EW algorithm obtains improvements of up to 17% when compared to the EW algorithm.

Keywords: Capacitated minimum spanning trees, network design, optimization

1 Introduction

We consider the well-known *Capacitated Minimum Spanning Tree* (CMST) problem. Given a set of nodes, each associated with a positive number denoting the traffic to be routed to the central node (root), the CMST problem asks for a minimum cost spanning tree, spanning all nodes, such that the amount of traffic routed from a subtree connected to the root does not exceed the given capacity constraint k. The cost of a tree is defined to be the sum of cost of the edges in that tree. The CMST problem can formally be defined as follows.

CMST: Consider a given undirected, vertex-weighted, complete graph $G = (V, E)$, where V is the set of nodes and E is the set of edges connecting the nodes, root node $r \in V$, and a positive integer k. Edges have positive costs associated with them. The CMST problem asks for a minimum cost spanning tree, with the sum of the vertex-weights of every subtree connected to r being at most k.

In the design of telecommunication networks, the CMST problem corresponds to the designing of a minimum cost tree network by installing expensive (fiber-optic) cables along its edges, with a capacity constraint k on the cable being used. Since the cable capacity is k, every subtree connected to the root can route a traffic of at most k to the root. Access networks are the ends or tails of networks. If local access represents most of the total network cost, it is critical to design low-cost local access networks. A CMST is a good choice for local access network design if the access network is required to be a tree. We refer the reader to Cahn's book [5] for more details on how the CMST problem represents a good design for local access network.

We use the terms nodes and vertices interchangeably. The decision version of the CMST problem is NP-complete [7, 11], even when vertices have equal weights and $k = 3$. However, the problem is polynomial-time solvable if all vertices have unit weights and $k = 2$ [7]. Even the geometric version of the problem, where the edge lengths are the Euclidean distance between the vertices, remains NP-complete.

The CMST problem has been extensively studied in Computer Science and Operations Research for the past 40 years. Numerous heuristics and exact algorithms have been proposed (see [3, 8, 16] for detailed surveys). The size of the problem instances that can be solved to optimality by exact procedures, in reasonable amount of time, is still far from the size of the real-life instances. Current best heuristics for the CMST problem, in terms of the quality of the solutions produced, are due to Amberg et al. [3], Sharaiha et al. [14], and Ahuja et al. [1]. A major problem with these heuristics is that their worst-case running-times are not necessarily polynomial, i.e., they could be exponential [10, 15]. Because of their high running-times, they may not be practical for solving large problem instances. The most popular and efficient algorithm for the CMST problem is due to Esau and Williams [6], with a worst-case running time of $O(n^2 \log n)$, where n is the total number of nodes. Even though there are numerous other heuristics that claim to outperform Esau-Williams (EW) algorithm, none of them do so without substantial increase in running time.

In this paper, we propose a modification to the EW algorithm for the CMST problem. We will

henceforth be referring to the modified version of the EW algorithm as the modified EW (MEW) algorithm. Theoretically, the worst-case running time of the MEW algorithm is same as that of the EW algorithm, i.e., $O(n^2 \log n)$. However, in practice, the running time of MEW algorithm would be constant $d \leq 21$ times than that of the EW algorithm. In terms of the quality (cost) of the solutions produced, the MEW algorithm always outperforms or matches the EW algorithm. Experimental results on benchmark instances indicate that the MEW algorithm produces better results 67% of the time, when compared to the EW algorithm. There are instances for which we obtain improvements of 17%. For the most difficult set of test instances, in which the root vertex is placed in the corner, the MEW algorithm produces better results for 30% of the benchmark instances, when compared to the EW algorithm.

2 Overview of the EW Algorithm

Kruskal's algorithm [9] for constructing a minimum spanning tree (MST) starts with n subtrees, each containing just a node, and starts merging these subtrees until there is only one tree left. Merging of subtrees is based on the proximity of subtrees. In each step, two closest subtrees are merged by connected them using an edge. The final tree obtained in this manner is guaranteed to be an MST. A slightly modified version of the Kruskal's algorithm can be used to find a feasible CMST for a given instance. One of the modifications required in the Kruskal's algorithm is to connect a growing subtree to the root vertex once the sum of the weights of the vertices in the subtree reaches the capacity k. In addition, we should not allow two subtrees whose sum of weights exceed k to merge since it would result in a subtree of weight greater than k. If there exists no two subtrees that can be merged, then all the remaining subtrees are connected to the root using direct edges.

The EW algorithm, instead of using proximity to merge two different subtrees, introduces the notion of "savings" to merge any two subtrees. While Kruskal's algorithm merges two closest subtrees at any point in time, the EW algorithm computes the savings involved in merging any two subtrees, and connects the two subtrees that produces the maximum savings (most negative value). Let `dist` be a $n \times n$ matrix, with $\texttt{dist}(i, j)$ denoting the distance (or cost of the edge) between nodes i and j. Kruskal's algorithm, in each iteration, chooses the smallest entry from the `dist` matrix, and connects the two nodes (merges the two subtrees in which the corresponding nodes are present) corresponding to that entry if no cycle is formed. The EW algorithm, on the other hand, computes a $n \times n$ `savings` matrix using the following formula:

$$\texttt{savings}(i, j) = \texttt{dist}(i, j) - \texttt{dist}(i, r)$$

where r is the root vertex. The EW algorithm works exactly the same way as the Kruskal's algorithm, except on the `savings` matrix instead of the `dist` matrix. Unlike Kruskal's algorithm, a merge is attractive only if the savings are negative. Subtrees i and j, which produce maximum

savings (most negative value) and whose sum of vertex weights is less than or equal to k are merged, and the `savings` matrix is recomputed for the next step.

A well-known major problem with the EW algorithm is when all the subtrees formed thus far have weight just over $k/2$, making it impossible to merge any two of them [13]. In this case, the EW algorithm will connect each subtree directly to the root. This could result in almost twice as many subtrees compared to an optimal solution. Fig. 1 depicts this pictorially for a sample scenario. While an optimal solution is of cost 2+, the solution produced by the EW algorithm will be of cost 3+.

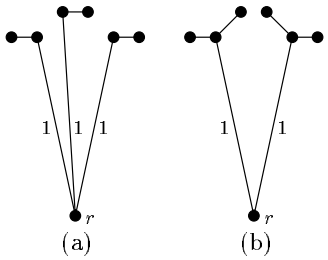


Figure 1: *Problem instance with* $k = 3$. (a) *Solution obtained by the EW algorithm with cost* 3+. (b) *Optimum solution with cost* 2+.

Consider Fig. 2(a) for the worst-case scenario, with 10,000 unit weight nodes, a root node, and $k = 100$. Let the cost of edges connecting the root node r to any other node be $M = 1,000,000$. Let the nodes be scattered in groups of 51 nodes with the cost of the edge connecting any two nodes in the same group being 1 and the cost of the edge connecting nodes from two different groups be $1 + \epsilon$, where $\epsilon = 1/M$. For this sample scenario, the EW algorithm will form subtrees of size 51 as shown in Fig. 2(b), while an optimal solution would form subtrees of size exactly $k = 100$ connected to r. Thus, the cost of the solution obtained by the EW algorithm will be approximately $M * 10,000/51 \approx 196M$, while the cost of an optimal solution is roughly $100M$. Generalizing this example, one can construct examples for which the EW algorithm will produce solutions of cost almost twice as much as an optimal solution.

3 The MEW Algorithm

3.1 Motivation

The main motivation to improve the EW algorithm arose from the fact that the EW algorithm could end up with a solution with twice the number of subtrees as compared to an optimal solution, which in turn could increase the cost of the obtained solution. Our initial idea was to reduce the number of almost-equal weight subtrees, especially subtrees of weight just over $k/2$, as this was the major cause of concern in the EW algorithm. To eliminate this problem, merging of subtrees should be done in a more controlled fashion. It could mean prioritizing the subtrees based on their weights so that at any point during the execution of the algorithm, subtrees of greater weight have

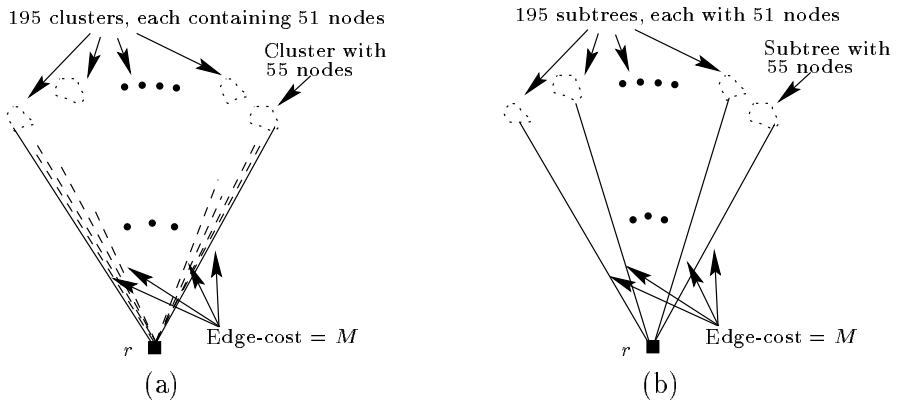


Figure 2: (a) *Problem instance with 10,000 unit weight nodes and* $k = 100$. (b) *Solution obtained by the EW algorithm with cost* $\approx 196M$.

an influence in the merging process. Giving a higher priority to subtrees of greater weight will allow subtrees of greater weight to grow bigger and bigger (up to to a weight of k), thereby reducing the possibility of almost-equal weight subtrees with weight just over $k/2$.

In what follows, we first present the reasoning the MEW algorithm, followed by the algorithm itself. There are two kinds of costs involved in constructing a CMST. We have the cost of connecting the non-root nodes together which we call the *subtree cost*, and the cost of connecting the subtrees, of weight at most k, to the root using a "radial spoke" which we call as the *spoke cost*. In fact, the cost of an optimal CMST can be bounded from below by two different quantities, namely the subtree cost lower bound (TLB), i.e., the MST cost, and the spoke lower bound (SLB), which is the sum of the radial distances from the root to all the nodes divided by k [2]. Kruskal's algorithm and its variants only look at the subtree cost, and can be viewed as being based on TLB. The EW algorithm tries to balance the subtree cost with the spoke cost, and therefore tries to use both TLB and SLB.

We try to improve this balance achieved, by making the following observation. The EW algorithm makes a decision to add an edge (i, j) to connect two subtrees (one containing i and the other containing j) without considering the sizes of the two subtrees being merged. In reality, only TLB is independent of the subtree size. On the other hand, since the SLB is proportional to the number of nodes in the subtree, more weight should be given to a subtree as it gets larger. In other words, when we add an edge to another subtree that takes us closer to the root, we save more when there are more nodes in this subtree than not.

The example in Fig. 3 illustrates this idea better. Let k be 10. Out of the several growing subtrees, let us focus on the following three subtrees: subtree i with weight 5, subtree j with weight 2, and subtree w with weight 1. Let the cost of the edge between w and j be c, and the cost of the edge between w and i be $c + \epsilon$, where $\epsilon << c$. Let the cost of the edge connecting i (or j) to root r be $s > c$, and the cost to connect w to r be $t < s$. According to the EW algorithm,

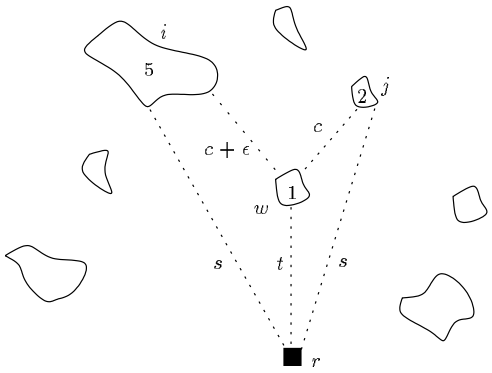


Figure 3:

- $\texttt{savings}(i, w) = \texttt{dist}(i, w) - \texttt{dist}(i, r) = (c + \epsilon) - s$,
- $\texttt{savings}(j, w) = \texttt{dist}(j, w) - \texttt{dist}(j, r) = c - s$.

Recall that merging of subtrees is attractive only when the savings is negative. Since the potential savings from connecting j to w is lesser than that of connecting i to w, the EW algorithm will choose to connect subtrees j and w. As per the MEW algorithm, the option of connecting i to w is considered depending on the value of a parameter δ (explained later in Section 3.2). Doing this could potentially prevent situations such as the one discussed in Section 2. We suggest the following modification to the EW algorithm to improve its performance.

3.2 Modified Savings Equation

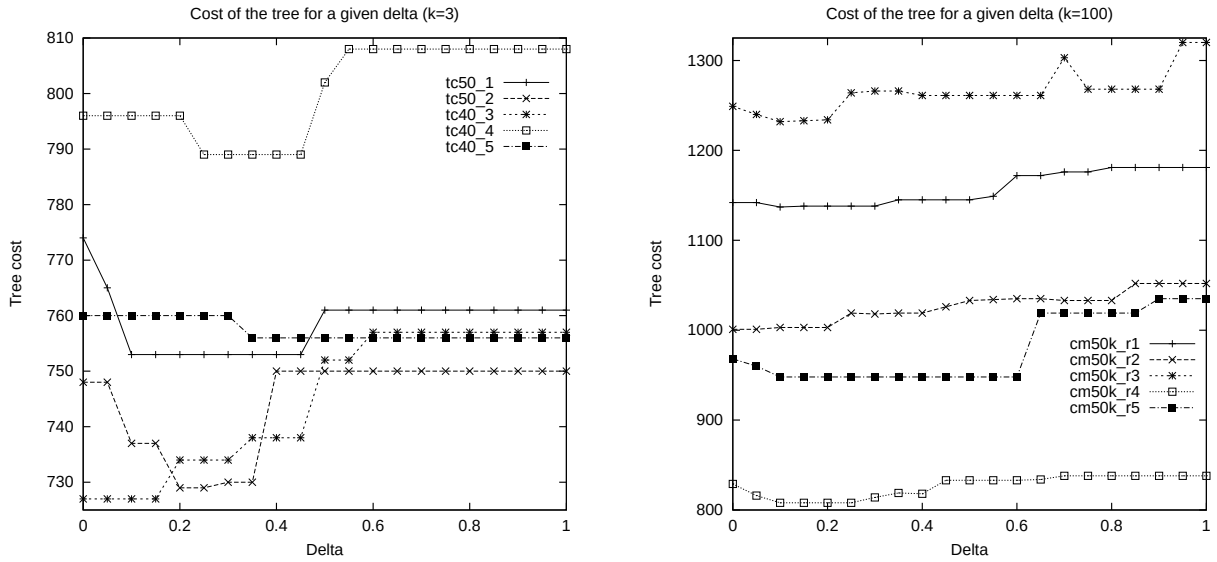
Let w_i denote the sum of the weights of the vertices in subtree i. We transformed the EW savings equation into the following weighted equation,

$$\texttt{savings}(i, j) = \Big(\texttt{dist}(i, j) - \texttt{dist}(i, r)\Big) \times w_i^{\delta}$$

where δ is a user-defined constant between 0.0 and 1.0. Note that when $\delta = 0.0$, the above equation is same as the EW algorithm. The rest of the EW algorithm is left unchanged. Since computation of w_i is part of the EW algorithm, the time to compute the weighted equation is same as that of computing the EW savings equation.

Our experiments showed that it is not easy to fix δ. In graphs with huge vertex weights and relatively small edge costs, fixing δ to some value could cause subtrees with more weight to completely dominate the savings value produced by the new savings equation. Similarly for graphs with huge edge costs and negligible vertex weights, fixing δ to any value might not produce the desired result. Thus, no matter how δ is chosen, one can construct a graph for which the choice of δ is wrong.

Preliminary tests on some benchmark instances showed that best solutions were obtained most often for a fixed δ value in the range 0 to 0.25. Fig. 4(a) shows the performance plots for five



(a) (b)

Figure 4: *Benchmark instances with unit vertex weights:* (a) *edges satisfying triangle inequality* (b) *edges not satisfying triangle inequality*

benchmark instances ($tc40_*.txt, k = 1, \ldots, 5$) obtained from OR-library [4] with unit weight vertices and $k = 3$. Fig. 4(b) shows the performance plots for five benchmark instances ($cm50_*.txt, k = 1, \ldots, 5$) obtained from OR-library with non-unit vertex weights given by $priz50.txt$ [4] and $k = 100$. After extensive testing on 105 benchmark instances (and numerous other randomly generated graphs), a clear pattern on δ value was observed. For the 105 benchmark instances, Table 1 shows the values of δ for which the low-cost solutions were obtained. Every instance was tested with varying δ values. We chose δ in fixed increments of 0.05.

δ	Number of best solutions
0.00	48
0.05-0.25	72
0.30-0.50	19
0.55-0.75	9
0.80-1.00	6

Table 1: *Number of low-cost solutions obtained for different values of δ.*

Despite the clear pattern on the value of δ for which the best solutions were obtained, we designed the MEW algorithm to run for constant d times, each time with a different δ value. This increases the worst-case running time only by a constant factor d (number of different δ values). We chose d to be 21, with values of $\delta = \{0.0, 0.05, 0.10, \ldots, 1.00\}$. Out of the 21 possible solutions obtained, the MEW algorithm returns the best solution. Even though the running time of the MEW algorithm is theoretically $O(n^2 \log n)$, in practice, for 21 values of δ, the running time will

be 21 times than that of the EW algorithm, regardless of the value of n. If the practical running time is of paramount concern, one can always choose to run the MEW algorithm for just 6 values of δ (say $\delta = \{0.0, 0.05, ..., 0.25\}$), which would make the running time to be 6 times than that of the EW algorithm, but the quality of the solution may be compromised.

As for as the problem instance in Fig. 2(a), for some δ value, the MEW algorithm, at some point during the execution of the MEW algorithm, will actually consider introducing edges between nodes in different clusters because of its weighted savings equation. This is unlike the EW algorithm in which $\delta = 0$, the edges between nodes in different clusters will never be considered during the course of the algorithm, thereby producing twice the number of subtrees compared to an optimal solution 2(b).

Theorem 3.1 *For a given CMST instance, the MEW algorithm always outperforms or matches the EW algorithm in terms of the cost of the solution obtained.*

Proof. The MEW algorithm with $\delta = 0.0$ is just the EW algorithm. The fact that we run the MEW algorithm for d different values of δ, one of which is 0.0, and output the best (low-cost) out of the d possible solutions completes the proof. ∎

4 Experimental Results

For comparison purposes, we evaluated the performance of the MEW algorithm on benchmark instances from OR-library [4]. For a given instance, we use n to denote the number of nodes.

Our test set included 105 benchmark instances. The test set was classified into three groups of problems. Our first two test groups are the *tc* and *te* groups (30 instances each) with unit vertex weights. Edges in both *tc* and *te* problems satisfy triangle inequality. In *tc* problems, the root was placed in the center of the vertex scatter and in *te* problems, the root was placed in the corner of the vertex scatter. Both *tc* and *te* groups contain 10 problems each, 5 of size $n = 40$ ($tc40\_*$ and $te40\_*, * = \{1, \ldots, 5\}$) and 5 of size $n = 80$ ($tc80\_*$ and $te80\_*, * = \{1, \ldots, 5\}$). Every problem of size $n = 40$ was tested with values of $k = \{3, 5, 10\}$ and every problem of size $n = 80$ was tested with values of $k = \{5, 10, 20\}$.

Test results for *tc* problems (Table 2) indicate that the MEW algorithm produces better solutions than the EW algorithm in 67% of the instances. On average, solutions obtained by the MEW were off by 3.1% when compared to the known lower bounds for the respective problems, while solutions obtained by the EW algorithm were off by 3.9%. As expected, test results for the tougher *te* problems (Table 3) were not as good as the ones obtained for *tc* problems. Our algorithm produced better solutions in only 30% of the *te* instances.

The third group of problems that we tested was the *cm* group (45 instances), with non-unit vertex weights. Unlike *tc* and *te* problems, edges in *cm* problems do not satisfy triangle inequality. The *cm* group of problems consists of 15 problems, 5 of size $n = 50$ ($cm50\_r*, * = \{1, \ldots, 5\}$), 5 of size $n = 100$ ($cm100\_r*, * = \{1, \ldots, 5\}$), and 5 of size $n = 200$ ($cm200\_r*, * = \{1, \ldots, 5\}$). Every

cm problem was tested with values of $k = \{100, 200, 400\}$. Our test results for the *cm* problems (Table 4) indicate that the MEW algorithm produces better solutions in 67% of the test instances. The MEW algorithm obtains improvements of up to 17% when compared to the EW algorithm. On average, the MEW algorithm obtains an improvement of 1.6% when compared to the EW algorithm.

			Solutions			% Gap	
Problem	n	k	EW	MEW	LB†	EW	MEW
tc40_1	41	3	774	**753**	742*	4.31	1.48
tc40_2	41	3	748	**729**	717	4.32	1.67
tc40_3	41	3	727	727	716	1.54	1.54
tc40_4	41	3	796	**789**	775	2.71	1.81
tc40_5	41	3	760	**756**	741*	2.56	2.02
tc40_1	41	5	597	**595**	586*	1.88	1.54
tc40_2	41	5	588	**583**	578	1.73	0.87
tc40_3	41	5	607	607	577*	5.20	5.20
tc40_4	41	5	639	**623**	617*	3.57	0.97
tc40_5	41	5	615	615	600	2.50	2.50
tc40_1	41	10	506	506	498*	1.61	1.61
tc40_2	41	10	504	**502**	490	2.86	2.45
tc40_3	41	10	508	508	500*	1.60	1.60
tc40_4	41	10	530	530	512*	3.52	3.52
tc40_5	41	10	504	504	504*	0.00	0.00
tc80_1	81	5	1184	**1182**	1094	8.23	8.04
tc80_2	81	5	1153	1153	1090	5.78	5.78
tc80_3	81	5	1144	**1127**	1067	7.22	5.62
tc80_4	81	5	1146	**1136**	1070	7.10	6.17
tc80_5	81	5	1367	**1352**	1268	7.81	6.62
tc80_1	81	10	948	**933**	878	7.97	6.26
tc80_2	81	10	929	929	875	6.17	6.17
tc80_3	81	10	908	**904**	869	4.49	4.03
tc80_4	81	10	921	**914**	863	6.72	5.91
tc80_5	81	10	1025	1025	998	2.71	2.71
tc80_1	81	20	862	**842**	834*	3.36	0.96
tc80_2	81	20	834	834	820*	1.71	1.71
tc80_3	81	20	846	**836**	828*	2.17	0.97
tc80_4	81	20	830	830	820*	1.22	1.22
tc80_5	81	20	948	**936**	916*	3.49	2.18

* Optimum solution † Data obtained from [12]

Table 2: *Computational results for the tc problems with unit vertex weights (n - number of nodes, k - capacity constraint, LB - Lower bound).*

5 Conclusion

We presented the MEW algorithm, a modified version of the popular EW algorithm, for finding CMSTs. In the MEW algorithm, the original EW savings equation is replaced with a weighted savings equation. Our proposed modification may help in reducing the number of subtrees connected to the root vertex, which will result in a lower cost solution. Our experiments show that, in majority of the instances, the MEW algorithm outperforms the EW algorithm in terms of the cost

			Solutions			% Gap	
Problem	n	k	EW	MEW	LB†	EW	MEW
te40_1	41	3	1208	1208	1190	1.51	1.51
te40_2	41	3	1140	1140	1103	3.35	3.35
te40_3	41	3	1148	**1139**	1115*	2.96	2.15
te40_4	41	3	1153	1153	1132	1.86	1.86
te40_5	41	3	1139	**1124**	1104	3.17	1.81
te40_1	41	5	867	867	830	4.46	4.46
te40_2	41	5	822	822	792	3.79	3.79
te40_3	41	5	820	820	797	2.89	2.89
te40_4	41	5	870	**867**	814	6.88	6.51
te40_5	41	5	812	**805**	784	3.57	2.68
te40_1	41	10	639	639	596	7.21	7.21
te40_2	41	10	607	607	573	5.93	5.93
te40_3	41	10	587	587	568	3.35	3.35
te40_4	41	10	600	600	596	0.67	0.67
te40_5	41	10	593	593	572*	3.67	3.67
te80_1	81	5	2618	2618	2531	3.44	3.44
te80_2	81	5	2613	2613	2522	3.61	3.61
te80_3	81	5	2707	**2701**	2593	4.40	4.17
te80_4	81	5	2639	**2633**	2539	3.94	3.70
te80_5	81	5	2578	2578	2458	4.88	4.88
te80_1	81	10	1716	1716	1631	5.21	5.21
te80_2	81	10	1713	1713	1602	6.93	6.93
te80_3	81	10	1781	1781	1660	7.29	7.29
te80_4	81	10	1792	**1691**	1614	11.03	4.77
te80_5	81	10	1708	1708	1586	7.69	7.69
te80_1	81	20	1308	1308	1256	4.14	4.14
te80_1	81	20	1292	1292	1201	7.58	7.58
te80_1	81	20	1342	**1341**	1257	6.76	6.68
te80_1	81	20	1372	1372	1247	10.02	10.02
te80_1	81	20	1290	**1289**	1231	4.79	4.71

* Optimum solution † Data obtained from [12]

Table 3: *Computational results for the te problems with unit vertex weights (n - number of nodes, k - capacity constraint, LB - Lower bound).*

of the solutions obtained. Overall, the MEW algorithm is guaranteed to at least match the costs of the solutions obtained by the EW algorithm. The effective running time of the MEW algorithm is only a constant $d \leq 21$ times than that of the EW algorithm.

References

[1] R.K. Ahuja, J.B. Orlin and D. Sharma, A composite neighborhood search algorithm for the capacitated minimum spanning tree problem, Oper. Res. Letters 31 (2001) 85-94.

[2] K. Altinkemer and B. Gavish, Heuristics with constant error guarantees for the design of tree networks, Management Science 34 (1988) 331-341.

[3] A. Amberg, W. Domschke and S. Voß, Capacitated minimum spanning trees: Algorithms using intelligent search, Combinatorial Optimization: Theory and Practice 1 (1996) 9-39.

[4] J.E. Beasley, OR-Library, http://www.ms.ic.ac.uk/info.html.

[5] R.S. Cahn, Wide Area Network Design: Concepts and Tools for Optimization, Morgan Kaufmann, 1998.

[6] L.R. Esau and K.C. Williams, On teleprocessing system design, IBM Systems Journal 5 (1966) 142-147.

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

[8] R. Jothi and B. Raghavachari, Approximation algorithms for the capacitated minimum spanning tree problem and its variants in network design, in Proc. International Colloquium on Automata, Languages and Programming (2004) 805-818.

[9] J.B. Kruskal Jr., On the shortest spanning subtree of a graph and the traveling salesman problem, Proc. Amer. Math. Soc. 7 (1956) 48-50.

[10] J. Orlin, MIT, Personal communication.

[11] C.H. Papadimitriou, The complexity of the capacitated tree problem, Networks 8 (1978) 217-230.

[12] R. Patterson, H. Pirkul and E. Rolland, A memory adaptive reasoning technique for solving the capacitated minimum spanning tree problem, Journal of Heuristics 5 (July 1999) 159-180.

[13] T.G. Robertazzi, Planning Telecommunication Networks, Wiley-IEEE Press, 1999.

[14] Y.M. Sharaiha, M. Gendreau, G. Laporte and I.H. Osman, A tabu search algorithm for the capacitated shortest spanning tree problem, Networks 29 (1997) 161-171.

[15] D. Sharma, Univ. of Michigan-Ann Arbor, Personal communication.

[16] S. Voß, Capacitated minimum spanning trees, In C.A. Floudas and P.M. Pardalos (eds.), Encyclopedia of Optimization 1 (2001) 225-235.

Problem	n	k	EW	MEW	% Gap
cm50_r1	50	100	1142	**1137**	-0.44
cm50_r2	50	100	1001	1001	0.00
cm50_r3	50	100	1249	**1232**	-1.36
cm50_r4	50	100	829	**808**	-2.53
cm50_r5	50	100	968	**948**	-2.07
cm50_r1	50	200	717	**713**	-0.56
cm50_r2	50	200	658	658	0.00
cm50_r3	50	200	739	739	0.00
cm50_r4	50	200	571	**568**	-0.53
cm50_r5	50	200	638	638	0.00
cm50_r1	50	400	518	**514**	-0.77
cm50_r2	50	400	545	**538**	-1.28
cm50_r3	50	400	541	541	0.00
cm50_r4	50	400	491	**490**	-0.20
cm50_r5	50	400	513	**508**	-0.97
cm100_r1	100	100	700	**686**	-2.00
cm100_r2	100	100	766	**759**	-0.91
cm100_r3	100	100	749	**721**	-3.74
cm100_r4	100	100	562	**529**	-5.87
cm100_r5	100	100	562	**537**	-4.45
cm100_r1	100	200	315	**304**	-3.49
cm100_r2	100	200	335	**316**	-5.67
cm100_r3	100	200	302	**299**	-0.99
cm100_r4	100	200	278	278	0.00
cm100_r5	100	200	308	**254**	**-17.53**
cm100_r1	100	400	201	201	0.00
cm100_r2	100	400	192	**188**	-2.08
cm100_r3	100	400	183	183	0.00
cm100_r4	100	400	194	**193**	-0.52
cm100_r5	100	400	215	**205**	-4.65
cm200_r1	200	100	1332	1332	0.00
cm200_r2	200	100	1703	**1662**	-2.41
cm200_r3	200	100	1682	**1681**	-0.06
cm200_r4	200	100	1304	1304	0.00
cm200_r5	200	100	1306	**1305**	-0.08
cm200_r1	200	200	500	500	0.00
cm200_r2	200	200	621	621	0.00
cm200_r3	200	200	702	**693**	-1.28
cm200_r4	200	200	517	**504**	-2.51
cm200_r5	200	200	504	**501**	-0.60
cm200_r1	200	400	301	301	0.00
cm200_r2	200	400	329	329	0.00
cm200_r3	200	400	393	**392**	-0.25
cm200_r4	200	400	308	**304**	-1.30
cm200_r5	200	400	329	329	0.00

Table 4: *Computational results for the cm problems with non-unit vertex weights (n - number of nodes, k - capacity constraint).*