Finding Minimum Congestion Spanning Trees

Renato F. Werneck
and
João C. Setubal
Institute of Computing, University of Campinas, Brazil

Given a weighted graph $G = (V, E)$, a positive integer k, and a penalty function w_p, we want to find k spanning trees on G, not necessarily disjoint, of minimum total weight, such that the weight of each edge is subject to a penalty given by w_p if it belongs to more than one tree. The objective function to be minimized is $\sum_{e \in E} W_e(i_e)$, where i_e is the number of times edge e appears in the solution and $W_e(i_e) = i_e w_p(e, i_e)$ is the aggregate cost of using edge e i_e times. For the case when W_e is weakly convex, which should have wide application in congestion problems, we present a polynomial time algorithm; the algorithm's complexity is quadratic in k. We also present two heuristics with complexity linear in k. In an experimental study we show that these heuristics are much faster than the exact algorithm also in practice. These experiments present a diverse combination of input families (four), varying k (up to 1000), and penalty functions (two). In most inputs tested the solutions given by the heuristics were within 1% of optimal or much better, especially for large k. The worst quality observed was 3.2% of optimal.

1. INTRODUCTION

In this paper we study a combinatorial *congestion problem.* We define a congestion problem as an allocation problem where each "resource" may be shared by any number of "users", but where any actual sharing implies an increase in the cost paid by each user. A typical congestion problem is traffic congestion on a road, in which each driver on that road suffers delays. Given an undirected weighted graph

This work was partially supported by FAPESP undergraduate fellowship 98/13423-5 (R.F.W.); CNPq fellowship 301169/97-7 and Pronex project 107/97 (J.C.S.).
An earlier version of this paper appeared in the *Proceedings of the Third Workshop on Algorithm Engineering,* London, 1999.
Name: Renato F. Werneck (rwerneck@inf.puc-rio.br).
Affiliation: Department of Informatics, Pontifícia Universidade Católica do Rio de Janeiro.
Address: Rua Marquês de São Vicente, 225 RDC, Rio de Janeiro, RJ 22453-900, Brazil.
Name: João Carlos Setubal (setubal@ic.unicamp.br) (to whom correspondence should be addressed).
Affiliation: Institute of Computing, Universidade Estadual de Campinas.
Address: C.P. 6176, Campinas, SP 13083-970, Brazil.

$G = (V, E)$ with n vertices, m edges, and edge weight function w, we can define "congestion variants" for several of the well-known graph optimization problems. In this paper we are interested in one such variant that is related to the minimum spanning tree problem.

Let k be a positive integer. Our aim will be to find a set $\mathcal{T}$ of not necessarily disjoint k spanning trees $T_1, T_2, \ldots, T_k$ of a graph G, subject to certain conditions. Let i_e be the number of times that edge $e \in E$ appears in (or "is used by") the trees in $\mathcal{T}$. The cost for each use of an edge e, instead of being a constant $w(e)$, is some function $w_p(e, i_e)$ of $w(e)$ and the total number of uses i_e of that edge (we call w_p the *penalty function*). The total cost of a solution $\mathcal{T}$ could then be defined as $\sum_{e \in E} w_p(e, i_e)$. However, here we are concerned with the *aggregate cost* for an edge e, which is defined as $W_e(i_e) = i_e w_p(e, i_e)$. This aggregate cost takes into account not only the mean cost per user of an edge [the value of $w_p(e, i_e)$], but also the sum of all such costs over all chosen trees. We use this aggregate formulation because we believe it more clearly models global congestion situations.

The problem studied in this work can now be stated as follows:

> Given G, k, and w_p, find k spanning trees $T_1, T_2, \ldots, T_k$ of G, not necessarily disjoint, such that $\sum_{e \in E} W_e(i_e)$ is minimum.

If $w_p(e, 1) = w(e)$ and $w_p(e, i_e) = \infty$ for all $i_e > 1$, this is the problem of finding k *disjoint* spanning trees of minimum total cost, which we reference here as the kMSTd problem. We call the general problem the *minimum congestion k-spanning trees problem,* which we abbreviate by kMSTc. To our knowledge the general kMSTc problem has not been studied before. However, disjoint-trees versions of it are well-known. Nash-Williams [1961] and Tutte [1961] have studied the unweighted case. Roskind and Tarjan [1985], building on work of Edmonds [1965a, 1965b], presented a polynomial-time algorithm for the weighted case.

We present here results for the particular case where the aggregate cost function W_e is weakly convex; that is, when

$$W_e(i+1) - W_e(i) \geq W_e(i) - W_e(i-1) \tag{1}$$

for all $i \geq 1$ (we will refer to this as *the weak convexity condition*). Note that this condition implies that W_e grows at least linearly with i_e. (This condition also implies that $w_p(e, i)$ is nondecreasing, i.e. $w_p(e, i+1) \geq w_p(e, i)$ for all i; but the converse is not true.)

The reason why our results concern only this particular case will become clear in the next section. However, note that for congestion situations this case should have wide application. In many practical cases the penalty function grows much faster than linearly. For example, the load function for Internet traffic used by Fortz and Thorup [2000], which they claim is based on recommendations made by the company Cisco, grows superlinearly.

The kMSTc problem is interesting in its own right. However, our main interest in this paper is to explore it as an example of the following type of situation: A problem for which a polynomial-time algorithm exists, but for which, *in practice,* it is better to use heuristics. Heuristics are commonly encountered and vastly studied in the realm of NP-complete problems. In the case of problems in class P, such results are rarer. Examples from the literature are approximation algorithms for the graph matching problem [Williamson and Goemans 1996] and for the multicommodity flow problem [Leighton et al. 1995].

The results we present are as follows. First we show that the kMSTc problem under the weak convexity condition can be solved exactly in polynomial time. Then we present two algorithms for which we have no guarantees for solution quality, and hence are heuristics. We show that their theoretical running times are better than the running time of the exact algorithm. A summary of theoretical results is shown in Table 1. (Note that our theoretical results also hold if the objective function is $\sum_{e \in E} w_p(e, i_e)$, with w_p weakly convex.) We then present an experimental study that shows that the heuristics are indeed much faster than the exact algorithm (for the graph instances tested), while finding solutions very close to the optimum.

Table 1. Summary of theoretical results for the kMSTc problem.

algorithm	running time
exact	$O(m \log m + k^2 n^2)$
A-Prim	$O(k(m + n \log n))$
A-Kruskal	$O(m \log m + k(m + n \log n))$
B	$O(m \log m + kn(\log m + \alpha(2n, n) \log n))$

2. AN EXACT ALGORITHM

The exact algorithm is a simple reduction of our problem to the weighted disjoint-trees problem. Our algorithm therefore is entirely based on Roskind and Tarjan's algorithm (hence-forward called RT). Our reduction assumes that the RT algorithm can handle parallel edges. This is an easy extension of the algorithm described in [Roskind and Tarjan 1985].

Here is how the reduction works. Let the *incremental cost* $c(e, i_e + 1)$ of edge e be

$$c(e, i_e + 1) = W_e(i_e + 1) - W_e(i_e). \tag{2}$$

The value of $c(e, i_e + 1)$ represents the increase in the solution's value by the inclusion of edge e in the $(i_e + 1)$-th tree, and this function, given the weak convexity condition, is nondecreasing.

We will now reduce an instance $I = \langle G, k, w_p \rangle$ of the kMSTc problem to an instance $I' = \langle G', k, w' \rangle$ of the kMSTd problem. Given an instance of the kMSTc problem create a graph $G' = (V, E')$ in the following way: for each edge $e = (u, v) \in E$, create k parallel edges $e_1, e_2, \ldots, e_k \in E'$ (all of them between vertices u and v) such that $w'(e_j) = c(e, j)$, $1 \le j \le k$. Note that if G is connected, then G' will certainly contain at least k disjoint spanning trees. We remark that using parallel edges with incremental costs is a standard way of making such reductions; see for example [Ahuja et al. 1993, section 14.3].

We now claim that given a solution $\mathcal{T}'$ for I', we can obtain a solution $\mathcal{T}$ for I and vice versa. First note that, given a pair of vertices (u, v) and the edges $e_1, e_2, \ldots, e_k$ between them in I', an edge e_j can belong to $\mathcal{T}'$ if and only if every edge $e_l, 1 \leq l < j$, is also part of $\mathcal{T}'$.[1] This is so because $w'(e_l)$ is never greater than $w'(e_j)$, due to the nondecreasing behavior of $c(e, i_e)$. (And this explains why the weak convexity condition is crucial.) Furthermore, all these edges belong to different trees (to avoid cycles and because the trees must be disjoint). Based on this we can establish a one-to-one correspondence between a tree in $\mathcal{T}'$ and a tree in $\mathcal{T}$. If in $\mathcal{T}'$, for a given pair (u, v), all edges up to e_j are used, this means that $i_{(u,v)}$ in $\mathcal{T}$ will be j (and vice versa). The total cost of these edges in $\mathcal{T}$ or in $\mathcal{T}'$ is the same and that proves the claim.

We now very briefly describe the RT algorithm. It starts by sorting the edges in nondecreasing order by weight and creating a set $\mathcal{F}$ of k forests $F_1, F_2, ..., F_k$, each with n vertices and no edges. The algorithm executes the following augmenting step for each edge $e \in G$, in order: find an F_i such that e can be inserted in F_i without creating a cycle (with the insertion possibly causing a rearrangement of edges among the forests in $\mathcal{F}$); if such an F_i cannot be found, discard e. The algorithm stops when all forests become trees. The search for a valid F_i in each augmenting step is done by a labeling process.

Roskind and Tarjan show that this algorithm takes $O(m \log m + k^2 n^2)$ time. A direct application of the reduction above gives for our algorithm a running time of $O(km \log km + k^2 n^2)$ and a storage requirement of km edges. We now show that it is not necessary to store that many edges and that our algorithm can be made to run in the same time bound as the RT algorithm.

Given the input graph G, the reduction gives a clear rule for determining all km edges of G'. This means that we may keep G' implicit, and generate each copy of an edge from G on the fly. In addition, note that as soon as the algorithm decides to discard an edge e_j, all other unexamined edges $e_l, l > j$, between the same pair of vertices, can be discarded as well. This means that we do not even have to sort km values. We can either discard an edge and all remaining parallel edges, or we use edge e_j and insert its next copy (e_{j+1}) with its weight updated into the data structure that contains the edges. The time bound depends on the data structure used to represent the edge list. Since this list must also be updated, we need a priority queue. We assume that a binary heap is used.

Initialization of the algorithm consists of building the heap $[O(m)$ time$]$ and creating the forests $[O(kn)]$. Selecting an edge from the heap costs $O(1)$. Once an edge is processed, we may need to discard it or replace it with the next copy. In terms of heap operations, this means deleting or updating an element, respectively. Both require $O(\log m)$ time.

As in the original algorithm, the labeling step is executed $O(kn)$ times. Each execution requires $O(kn)$ operations, in addition to the extra $O(\log m)$ time demanded by heap operations. There are up to $O(m)$ steps in which applying the labeling algorithm is not necessary, since the ends of the edge being examined belong to the same special subset ("clump", in RT's terminology) of vertices. Each of these steps requires $O(\log m)$ time to delete the edge from the heap. The overall time bound of the algorithm implemented in this way is $O(m \log m + k^2 n^2)$, which is

[1]This may not be true if there are parallel edges with equal incremental cost. But in this case it is easy to obtain a solution $\mathcal{T}''$ equivalent to $\mathcal{T}'$ in which edges of equal incremental cost have been rearranged so that the statement is true.

an improvement over the direct application of the reduction and is the same time bound of the RT algorithm. We have thus established the following result:

THEOREM 1. *The kMSTc problem under the weak convexity condition can be solved in* $O(m \log m + k^2 n^2)$ *time.*

3. HEURISTICS

In this section we present two heuristics for the kMSTc problem. These heuristics were developed before we knew the complexity of the problem, and were thus contemplating the possibility that it might be NP-complete. Even though it turned out that the problem is polynomially solvable under the weak convexity condition, the heuristics proposed are intuitive, asymptotically faster than the exact algorithm, and simple to implement.

3.1 Heuristic A: Tree-Greedy

This is a greedy heuristic, in that it computes each tree T_i based on all previously computed trees T_j, $1 \leq j < i$. We start with graph $G_1 = G$, and compute T_1 as its minimum spanning tree. We then update the weights of edges that are in T_1. The updated weight of one such edge e will be $c(e, 2) = 2w_p(e, 2) - w_p(e, 1)$, its incremental cost as defined in (2). Graph G_2 is the same as G_1, but with edge weights updated as described. The next tree, T_2, will be the minimum spanning tree of G_2 and so on, until we have obtained k trees.

Any minimum spanning tree algorithm can be used to implement this heuristic. We analyze two possibilities: using Prim's algorithm and Kruskal's algorithm [Cormen et al. 1990].

Implementing Prim's algorithm using a Fibonacci heap makes it run in $O(m + n \log n)$ time. As Heuristic A executes this algorithm k times, its complexity is $O(k(m + n \log n))$.

Kruskal's algorithm requires a preprocessing edge-sorting step costing $O(m \log m)$. Using a union-find data structure the main loop costs $O(m\alpha(2n, n))$ time (using trees to implement sets, plus path compression and weighted union [Tarjan 1975; Cormen et al. 1990]) or $O(m + n \log n)$ (using linked lists and weighted union [Cormen et al. 1990]). The complexity of Kruskal's algorithm is thus dominated by the complexity of the preprocessing step. As Heuristic A requires k applications of the algorithm, its overall complexity would be $O(km \log m)$. However, Heuristic A allows us to execute Kruskal's algorithm more efficiently. After the i-th $(1 \leq i < k)$ execution of Kruskal's algorithm, the set of edges can be partitioned into two subsets, one (the first) containing the $n - 1$ edges of T_i, and another (the second) with the remaining edges. The latter is already sorted, since the incremental costs of its edges were not changed. For the next step, all we have to do is sort the first subset, merge it with the second, and apply Kruskal's algorithm again. This takes $O(m + n \log n)$ time. This observation frees us from sorting the complete set of edges in every step. The overall complexity of the heuristic is thus $O(m \log m + k(m + n \log n))$.

3.2 Heuristic B: Edge-Greedy

Heuristic B is greedy too, but "grows" the trees of the desired solution all together, in a fashion reminiscent of the exact algorithm. We start with k forests, each forest

F_i containing the n vertices of G and no edges. We pick the edge with the smallest incremental cost (and hence need a priority queue) and check whether it can be inserted into some forest without creating a cycle. If we find such a forest (which we call *valid*), the edge is inserted and its incremental cost is updated in the queue; if a forest is not found, the edge is discarded. This step is repeated until $k(n-1)$ edges are inserted into the forests, which by then are trees. There are at most $m-1$ steps in which a valid forest is not found, and $k(n-1)$ steps in which an edge is actually inserted. In each step, we must perform two kinds of operations:

—*Edge selection:* We must either remove an edge from the queue or update its weight. We can perform both operations in $O(\log m)$ time if we represent the priority queue as a binary heap.

—*Cycle checking:* The most efficient way to manage cycle-related information is employing a union-find data structure. Since each forest may be checked in every step, all cycle-related operations (considering the entire execution of the algorithm) cost $O((kn+m)k\alpha(2n,n))$ time.

All costs considered, the overall time bound of Heuristic B is $O((kn+m)(\log m + k\alpha(2n,n)))$. We will now show that this time bound can be substantially improved by using several different techniques.

3.2.1 *Monitoring Clusters.* The first idea is to use an extra union-find structure (in addition to the ones associated with forests) to represent *clusters*. Clusters are subsets of vertices that are in the same connected component in every forest F_i.

We start with n disjoint sets, each representing a vertex of G, and then proceed to the execution of the algorithm as previously described. If we test a candidate edge $e=(u,v)$ against every relevant forest and find that none is valid for insertion, we conclude that vertices u and v belong to the same cluster, and execute $union(u,v)$ in the extra union-find structure. The performance gain will be achieved if we verify whether the endpoints of a candidate edge belong to the same cluster before checking all forests in each step. If the vertices do belong to the same cluster, we simply discard the edge and proceed to the next one, avoiding a fruitless search.

The worst-case analysis of this approach is as follows: $k(n-1)$ successful searches, $O(n)$ unsuccessful searches (each resulting in a *union* operation), and $O(m)$ edges discarded without any search at all. Knowing that $O(\log m)$ operations are required to select an edge and that each search checks up to k forests, the overall time bound of this improved version of Heuristic B is $O(m\log m + kn(\log m + k\alpha(2n,n)))$.

3.2.2 *Discarding Trees.* Another idea is to test edges only against a relevant subset of the forests, those with more than one component (other forests can be discarded, since no edge can be added to them without creating cycles). To implement this, we keep the relevant forests (all of them, in the beginning of the algorithm) in a linked list. As soon the $(n-1)$-th edge is inserted into a forest F_i, we remove F_i from the list. This makes the algorithm faster, although its asymptotic time bound remains unchanged.

3.2.3 *Ordering.* For the running times presented so far, any forest scanning order can be used. One may even adopt different orders in each step. However, analyzing the forests in exactly the same order in every step yields a significant improvement.

It does not matter which order it is, as long as it is the same in every step.

To obtain the improvement we will need to compare connected components of the forests F_i. We will do so considering only the set of vertices in each component (*not* the edges). Suppose now that the fixed order in which forests are considered is $F_1, F_2, \ldots, F_k$. Then, the following is true:

THEOREM 2. *Any connected component in a forest F_i is a subset (not necessarily proper) of some connected component of F_{i-1}, for $1 < i \leq k$.*

PROOF. If there were a component C in F_i whose vertices were not in the same component of F_{i-1}, then there would be an edge e joining some two vertices of C that could be inserted in F_{i-1}. This contradicts the fact that edges are inserted in the first valid forest. □

The following corollary is immediate.

COROLLARY 1. *Let $e(u, v)$ be an edge of G and let F_i $(1 \leq i < k)$ be the first forest in which the insertion of e does not create a cycle (i.e., in which u and v are in different connected components). Then, e does not create a cycle in F_j, for every j such that $i < j \leq k$.*

As a consequence of this result, for each pair of vertices u, v, we can divide the set of forests into two subsets. In forests F_1 to F_{i-1}, u and v belong to the same connected component; in forests F_i to F_k, u and v are in different components. Our task is to find the first forest of the second subset. This can be done using binary search over the k forests. This reduces the overall complexity of the algorithm to $O(m \log m + kn(\log m + \alpha(2n, n) \log k))$. This complexity assumes that cluster monitoring (as described above) is used.

Applying the tree-discarding technique presented above is still possible. In fact, it becomes easier to implement when the forests are analyzed in a fixed order. It follows from Corollary 1 that a forest F_i $(1 < i \leq k)$ can become a tree only after every forest F_j $(1 \leq j < i)$ has already become a tree in previous steps. In other words, the subset of forests which are trees is either empty or can be expressed as $\{F_1, F_2, \ldots, F_s\}$, for some $s \leq k$. Thus, when a forest F_i becomes a tree, all we have to do is set $s = i$ and restrict further searches to forests $F_{s+1}, F_{s+2}, \ldots, F_k$. Notice that the single variable s makes the linked list for the tree-discarding technique unnecessary.

3.2.4 *Indexed Search.* Even with all the improvements mentioned so far, the running time of Heuristic B has a $k \log k$ factor, while Heuristic A is linear in k. As seen above, the extra $\log k$ factor is due to the cost of searching for a valid forest to insert a given edge. We now show that this search can be done in $O(\log n)$ time.

For this we need the concept of *component-equivalence.* Two forests $F_i(V, E_i)$ and $F_j(V, E_j)$ are component-equivalent if and only if their vertices are partitioned into the same connected components (recall that by "same component" we mean components with the same set of vertices, but not necessarily the same set of edges). With this definition we can now prove the following result.

THEOREM 3. *In any step of the algorithm, there are no more than* $\min(n, k)$ *sets of component-equivalent forests.*

PROOF. This is trivial if $k \leq n$, since k is the number of forests. Therefore, let us assume that $k > n$. An immediate consequence of Theorem 2 is that forests are always sorted in non-increasing order by number of edges. That is, given forests F_i and F_{i-1} ($1 < i \leq k$), either $|E_i| < |E_{i-1}|$ or $|E_i| = |E_{i-1}|$. They cannot be component-equivalent if $|E_i| < |E_{i-1}|$, since the number of edges is different. On the other hand, if $|E_i| = |E_{i-1}|$, Theorem 2 guarantees the component-equivalence of F_{i-1} and F_i, which means they belong to the same set. Thus the number of sets is equal to the number of strict inequalities. Since a forest with n vertices may have no more than $n-1$ edges, there are at most n different forest edge set cardinalities (from 0 to $n-1$) in a given step of the algorithm. Hence, n is the maximum number of different sets of component-equivalent forests if $k > n$. □

As described above, in every step of the algorithm we must look for a valid forest F_i for which i is minimum. As explained in Section 3.2.3 we can find this forest by performing binary search on the list of forests. Using Theorem 3, we show that there is a way of finding the same forest independent of k.

Let S_l ($0 \leq l \leq n-1$) be the set of component-equivalent forests with l edges. For a given edge e, we must find the largest l such that e can be inserted into a forest of S_l without creating a cycle. The edge e can be inserted into any forest in S_l. We choose to insert it in the first forest in S_l according to the order in which forests are being considered; this choice makes the algorithm asymptotically faster, as will be seen below. To carry out this insertion, we must keep for each set S_l a reference to its first forest, making the list of component-equivalent sets act as an index to individual forests. To find the desired forest, we can perform a binary search on a vector of indices, which has size n, costing us $O(\log n)$. We call this index vector *cindex*. This is the general idea. However, there is a minor detail that we must consider. Since any set S_l may be empty in some steps of the algorithm, we must decide which forest should be referenced to by $cindex[l]$ when this happens. We adopted the following solution: make $cindex[l]$ represent the first forest with l edges *or fewer*. Hence, when there is no l-edge forest, we have $cindex[l] = cindex[l-1]$. With this solution, *cindex* must be managed as follows:

—*Initialization:* Set $cindex[l] = 1$, for $0 \leq l \leq n-1$.

—*Search:* Given an edge e, find, using binary search, the largest l such that e can be inserted in $cindex[l]$ and cannot be inserted in $cindex[l+1]$.

—*Update:* Let F_i be the forest found in the previous step and $|E_i|$ be the number of edges in it (after e is inserted). Since $|E_i|$ increased by one, we have to do something about $cindex[|E_i|-1]$. Set $cindex[|E_i|-1] = i+1$, which means that F_{i+1} replaces F_i as the first forest with $|E_i|-1$ edges or fewer.

Notice that the second phase (search) can be unsuccessful, because there may be, in some step of the algorithm, no i such that e can be inserted in F_i. When this happens, e can be removed from the queue.

We can now present the final analysis of Heuristic B. The only difference we have introduced is the number of comparisons made when searching for a forest to insert an edge. When *cindex* is used, $O(\log n)$ comparisons are required per search, as opposed to $O(\log k)$ without that auxiliary vector. In both cases, each comparison takes at most $O(\alpha(2n, n))$ time on average, and all comparisons on all trees take

$O(kn\alpha(2n,n)\log n)$ total time. Therefore, Heuristic B with cluster control and indexed search will run in $O(m\log m + kn(\log m + \alpha(2n,n)\log n))$ time.

3.3 Heuristics A and B may produce identical solutions

Let Heuristic A be the generic heuristic proposed in Section 3.1. Let Heuristic B be the generic one proposed in the beginning of Section 3.2 with the addition that edges are inserted using a fixed order for forests (as described in Section 3.2.3 or in Section 3.2.4). Under the assumption that the instance to be solved has incremental costs that are pairwise distinct, we have the following result:

THEOREM 4. *Heuristics A and B under the above conditions produce exactly the same trees.*

PROOF. Let k be the number of trees to be found. We will prove that the i-th $(1 \le i \le k)$ tree found by A is the same as the i-th tree completed by B (we assume that the order of trees produced by B is given by the step in which the last edge of each tree is inserted). We do an induction on i.

Basis: $i = 1$. The first tree found by Heuristic A is the minimum spanning tree of input graph G, which is unique because costs are pairwise distinct.[2] Heuristic B behaves like Kruskal's algorithm in this case and hence finds this same tree. (The heuristic may consider an edge $e_p, p > 1$, before considering an "original" edge f_1, but no edge $e_p, p > 1$, can be inserted in the first tree.)

Induction hypothesis: Assume that the theorem is valid for $i < j$, meaning that the first $j-1$ trees found by A and the first $j-1$ trees completed by B are the same.

Induction step: Assume for the purpose of contradiction that the j-th trees found by the heuristics are different from each other ($T_j^A \neq T_j^B$). Let $e_p = (u,v)$ be the first edge in increasing order of incremental cost that differs between the two trees. Consider the case where $e_p \in T_j^A$ but $e_p \notin T_j^B$. Then consider the unique path from u to v in T_j^B. Not all edges along this path are present in T_j^A, otherwise there would be a cycle in T_j^A. So there must be an edge f_q in T_j^B that is not in T_j^A, and $w(f_q) > w(e_p)$ (because e_p is the lightest edge not present in both trees). This means that f_q was inserted in T_j^B but e_p was not. This is a contradiction because e_p was available for B to choose (the induction hypothesis guarantees that it does not belong to any of the previous $j-1$ trees) and because of the greedy nature of B. The case where $e_p \in T_j^B$ but $e_p \notin T_j^A$ uses essentially the same argument. In this case we have to argue that e_p was available for A to choose, which is true because of the same reasons of the previous case. The fact that the edge f_q in the unique path from u to v in T_j^A has $w(f_q) > w(e_p)$ shows that T_j^A is not a minimum spanning tree for the available edges, which is again a contradiction. □

We emphasize that this result is valid only if we consider Heuristic B with the fixed-forest ordering requirement. If edges are inserted choosing forests in an arbitrary order at each step, Heuristic B may find a solution different from the one found by A. In particular Heuristic B will find an exact solution if at each step it guesses correctly the tree where the current edge should be inserted.

[2]We assume here, without loss of generality, that $w_p(e,1) = w(e)$ for all edges e in G.

4. AN EXPERIMENTAL STUDY

We have implemented and tested the exact algorithm, Heuristic A with Kruskal's algorithm, Heuristic B (with all improvements described in Section 3.2), and a random algorithm (see next paragraph). Heuristic A with Prim's algorithm and a binary heap was implemented, but preliminary testing showed that it was considerably slower than A-Kruskal, and hence it was not part of further testing.

The random algorithm builds k trees, and for each tree it selects $(n-1)$ edges randomly, using a union-find data structure to avoid cycle-creating edges. This algorithm was implemented so that the proposed heuristics could be judged not only on how close they get to the optimal solution but also how much better they are with respect to a randomly found solution.

The implementation of the exact algorithm contains the following practical improvement in the RT algorithm, motivated by our experience in developing Heuristic B. When trying to insert an edge into a forest, we try all forests (in the same fixed order) rather than invoke the labeling procedure after failing to insert the edge into forest F_1. This modification changes neither the correctness (for both the kMSTd and kMSTc problems) nor the asymptotic running time of the RT algorithm [checking all forests in each step costs $O(k\alpha(2n,n))$, which is much less than the $O(kn)$ time required by the labeling procedure]. We verified that the RT algorithm ran considerably faster with this modification.

4.1 Methodology

All programs were implemented in C++ and compiled with the GNU C Compiler using the `-O3` optimization flag. All tests were done on a DEC Alpha 600au with 1 GB of RAM. Every instance fit in memory, thus limiting I/O operations to reading the input graph. I/O operations were not included in timing measurements.

The implementations were tested on families of instances created with generators that we wrote and on publicly-available instances. Original edge weights are always integers, but they are implemented using the `double` data type (because one of the penalty functions changes integer weights into rational weights). The families and instances are as follows:

—complu, a complete graph with uniform distribution of pairwise distinct edge weights.

—complb, a complete graph with a biased distribution of pairwise distinct edge weights (the bias is explained below).

—hyperu, a hypercubic grid with uniform distribution of pairwise distinct edge weights.

—hyperb, a hypercubic grid with a biased distribution of pairwise distinct edge weights (the same bias as in complb, see below).

—random, a random connected graph with uniform distribution of edge weights.

—brazil58, an instance from TSPLIB, obtained from the TSPLIB website, http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html. This is a complete graph with $n = 58$ and weights given by euclidean distances.

—orlib, instances for the Steiner tree problem from OR-library [Beasley 1990], obtained at ftp://mscmga.ms.ic.ac.uk/pub/. We used instances from "problem set C", which are random graphs, with $n = 500$, and uniformly distributed weights in the interval [1,10].

The hypercubic grid generator produces toroidal grids of order l and dimension d. For $d = 2$ and $l = 5$, this is a 2-dimensional 5×5 rectangular grid with wrap-around. Such grids have l^d vertices and dl^d edges.

Weight bias is obtained by having all weights in the interval $[1, Y]$ present, where $Y = \min(m, 2(n-1))$, and choosing remaining weights (if any) in a random and uniform way from the interval $[Y+1, \texttt{max}]$, where `max` is a program parameter.

We used two penalty functions in our experiments:

—Γ, given by $w_p(e, i_e) = i_e w(e)$. This results in a quadratic objective function.

—Φ, given by $w_p(e, i_e) = a(i_e, k)w(e)$, where a is a piece-wise linear convex function whose vertices are given in Table 2. Note that this function depends on both i_e and k. It was adapted from a similar one appearing in [Fortz and Thorup 2000].

Table 2. The penalty function Φ. Values for a when x is between the values given below are linearly interpolated.

$x = (i_e - 1)/k$	$a(x)$
0	1
1/3	3
2/3	10
9/10	70
1	500

In reporting results we use the concept of solution *quality*. The quality of solution S given by algorithm X is simply the ratio between the cost of S and the optimal cost.

4.2 Results

4.2.1 *Penalty function* Γ. We first present results that give a general picture of the performance of the various implementations with respect to input family, increasing n, increasing m, increasing k, time, and solution quality. In this first batch of results, the penalty function used was Γ.

Table 3 compares the three implementations with respect to running time, and the two heuristics and the random algorithm with respect to solution quality, for family complu and $n = 100$ with varying k. For each value of k, 10 different instances were generated (with different seeds) and tested. Edge weights are between 1 and $n(n-1)/2 = 4950$ (since graphs in this family are complete and edge weights are pairwise distinct, this means that there is a one-to-one correspondence between integers in the interval $[1, 4950]$ and edge weights). For each value of k and each implementation, the mean time in seconds and its standard deviation are shown. In the case of qualities we show the maximum value (worst quality) over the 10 instances for heuristics A and B, and the minimum value (best quality) for the random algorithm. This minimum value was obtained running the random algorithm 5 times on each instance. Hence, for each value of k, the minimum value was obtained from 50 different runs.

Table 3. Results for family complu, $n = 100$.

	time (s)			quality		
k	exact	A	B	A	B	random
100	2.492 ± 0.160	0.087 ± 0.006	0.040 ± 0.000	1.000384	1.000391	4.723017
200	7.569 ± 0.582	0.147 ± 0.016	0.080 ± 0.000	1.000251	1.000240	4.383867
300	13.514 ± 1.736	0.225 ± 0.031	0.120 ± 0.000	1.000274	1.000275	4.150481
400	17.866 ± 2.166	0.310 ± 0.051	0.160 ± 0.000	1.000139	1.000139	4.017073
500	20.770 ± 2.281	0.410 ± 0.077	0.199 ± 0.003	1.000065	1.000065	3.943122

Table 4. Results for family complu, $k = 100$.

	time (s)			quality		
n	exact	A	B	A	B	random
100	2.425 ± 0.091	0.088 ± 0.006	0.040 ± 0.000	1.000324	1.000337	4.695782
200	13.388 ± 0.488	0.329 ± 0.014	0.104 ± 0.005	1.000331	1.000332	6.360605
300	33.885 ± 1.127	0.719 ± 0.040	0.194 ± 0.005	1.000259	1.000257	8.007295
400	63.860 ± 2.653	1.237 ± 0.058	0.305 ± 0.007	1.000417	1.000419	9.712935
500	102.656 ± 3.409	1.893 ± 0.071	0.423 ± 0.009	1.000237	1.000237	11.315531

Table 5. Results for family hyperb, $k = 100$.

	time (s)			quality		
n	exact	A	B	A	B	random
81	0.756 ± 0.065	0.010 ± 0.000	0.020 ± 0.000	1.000222	1.000228	56.618198
256	11.141 ± 0.379	0.058 ± 0.004	0.100 ± 0.000	1.000351	1.000351	41.939893
625	78.322 ± 1.855	0.178 ± 0.004	0.310 ± 0.000	1.000348	1.000346	29.331106
1296	378.012 ± 4.320	0.438 ± 0.004	0.797 ± 0.005	1.000360	1.000359	15.834247

Table 6. Results for family orlib, $k = 100$, $n = 500$.

	time (s)			quality		
m	exact	A	B	A	B	random
625	24.142 ± 0.453	0.070 ± 0.000	0.204 ± 0.005	1.000062	1.000077	1.117696
1000	44.610 ± 1.363	0.092 ± 0.004	0.218 ± 0.004	1.000225	1.000292	1.330837
2500	81.930 ± 1.092	0.180 ± 0.000	0.224 ± 0.005	1.000287	1.000391	1.641769
12500	91.720 ± 1.776	0.266 ± 0.014	0.252 ± 0.007	1.000007	1.000012	1.939782

Table 4 compares the three implementations with respect to running time, and the two heuristics and the random algorithm with respect to solution quality, for family complu and $k = 100$ with varying n. Number of instances, edge weights and number of runs of the random algorithm are similar to the previous table.

Table 5 presents results similar to Table 4, but the instances come from family hyperb. Here weights are in the interval $[1, 100000]$. The parameter values used in the generator were $d = 4$ and $l = 3, 4, 5, 6$.

Table 6 presents results from experiments with family orlib. We recall that in this case the number of vertices is fixed at $n = 500$ and that the topology is random. Additionally, the values in each row result from five different instances (rather than ten, as in the previous tables), because only five instances for each m value were available at the address given above.

Tables 3, 4, 5, and 6 make it clear how much faster the heuristics are with respect to the exact algorithm and how good their solutions are. In all these cases heuristic solution quality is within 0.042% of optimal. In addition the tables show that the random algorithm is significantly worse than the heuristics. The difference in the quality values obtained by the random algorithm in families hyperb and orlib shows that this algorithm is quite sensitive to the weight distribution, as one would expect. Family orlib in particular is the best-case family we have for a random algorithm: as noted above, edge weights are in the small interval $[1, 10]$. Even in such a case, the difference in solution quality between the heuristics and the random algorithm is clear. In family complu we restricted edge weights to the interval $[1, m]$ so that they would be the smallest possible while still being pairwise distinct, thereby presenting another "good" case for a random algorithm. Even such instances, however, offer too many choices for the random algorithm, and therefore the quality of its solutions is significantly worse than that for family orlib, as shown in Table 3.

To get an idea of how quality changes with different values of k we ran both Heuristic A and B on instance brazil58. The results (taking the worst value between A and B for each k) are shown in Figure 1. We see that the worst case occurs for $k = 5$, and that for large values of k quality tends to 1. The random algorithm in this experiment had the following behavior: for $k = 1$, its quality was 6.99; for larger values of k, the quality improved, reaching 2.02 for $k = 1000$.

Given the relatively high value for the quality when $k = 5$ in the previous experiment, we decided to test the heuristics for small values of k on six other specific instances from three different families. In addition to the brazil58 instance already described, the other instances were as follows:

—complu100: $n = 100$, weights in the interval $[1 \dots 495000]$.

—complb100: $n = 100$, weights in the interval $[1 \dots 495000]$.

—hyperu256: $n = 256$, $m = 1024$, weights in the interval $[1 \dots 102400]$.

—hyperb256: $n = 256$, $m = 1024$, weights in the interval $[1 \dots 102400]$.

—orlib625: $n = 500$, $m = 625$ (the sparsest instance in this set), which is instance c01 in the orlib repository .

—orlib12500: $n = 500$, $m = 12500$, which is instance c16 in the orlib repository.

The results can be seen in Figures 2 and 3. Each point of each curve represents the worst quality between heuristics A and B. We take the opportunity here to

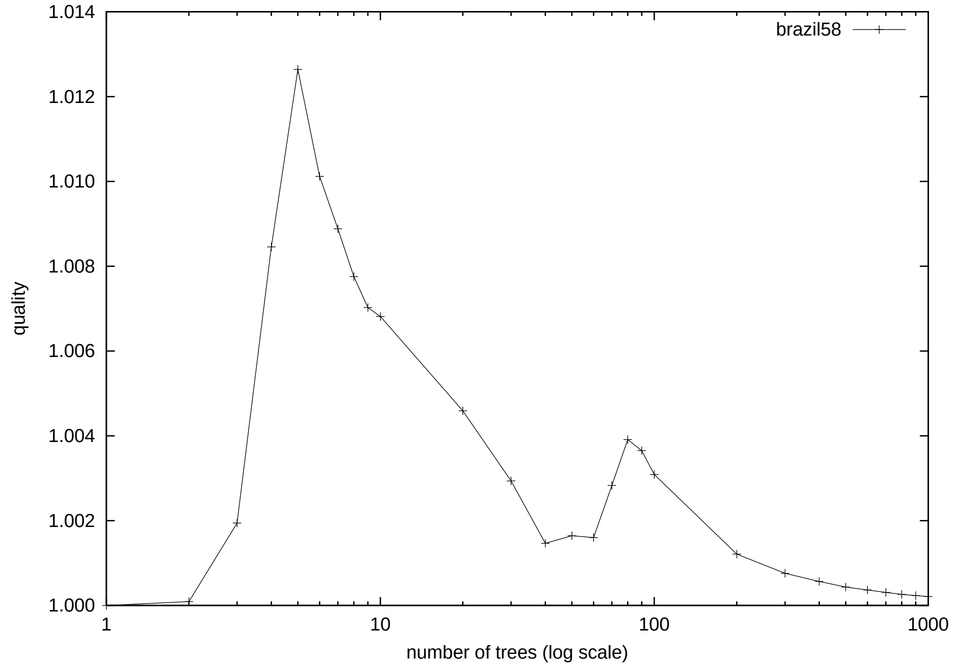

Fig. 1. Quality values on instance brazil58, penalty function Γ.

observe that performance of A and B with respect to quality was essentially the same in all tests we did (this can be clearly seen in Tables 3, 4, 5, and 6). This is a consequence of Theorem 4. The quality is not *exactly* the same because instances solved do not necessarily have incremental costs that are pairwise distinct.

Figures 2 and 3 show that quality values reach peaks in the interval $1 < k \leq 15$, and generally drop off for $k > 15$, for all instances. The peaks show that the heuristics obtained qualities within 1.3% of optimal, except on instance orlib12500 for $k = 5$, where quality was 2.8% of optimal.

4.2.2 *Penalty function* Φ. We now present a second batch of results obtained using function Φ. In reporting these results we concentrate on solution quality. Figures 4 and 5 are counterparts to Figures 2 and 3, respectively. The only change in the experimental set-up was the penalty function.

An analysis of these results shows first that quality values remain good. In instance hyperb256, we obtained the worst quality of all tests reported, which was 3.2% of optimal for $k = 2$. Another feature of these results is that for three of the instances (hyperb256, complb100, and orlib625) quality values do not tend to 1 after a certain value of k, as happened for function Γ. To gather further evidence of this phenomenon, we ran the heuristics on these three instances for k values up to 1000, and Figure 6 shows the results. There we see that after $k = 100$ or so quality values remain almost constant, and this constant in each case is larger than 1 (quality values are still quite good, being within 2.5% of optimal or better). We do not know the reason for this phenomenon but conjecture that it has to do

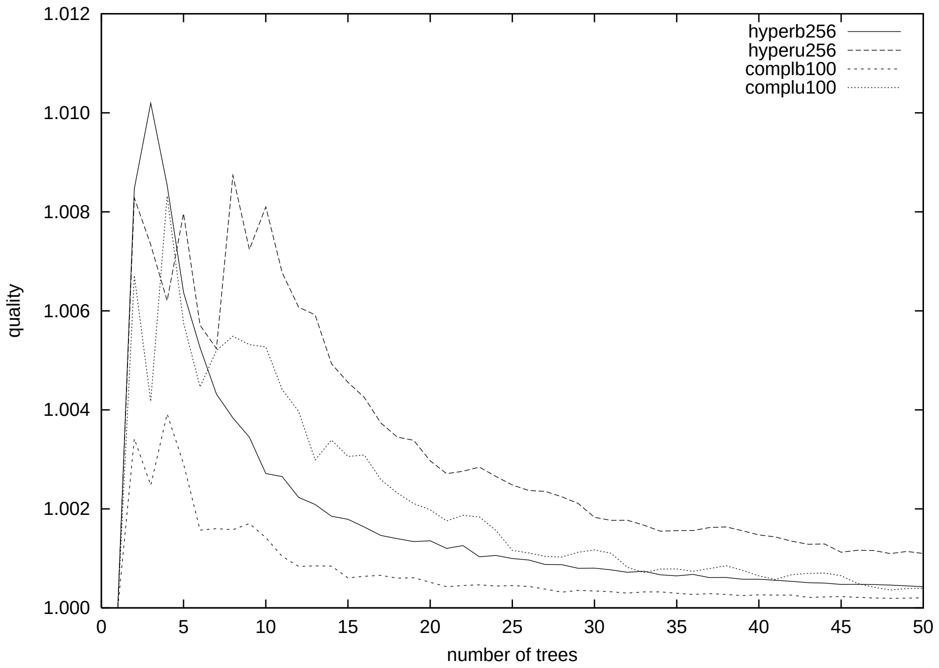

Fig. 2. Quality values for small values of k, first batch, penalty function Γ.

with the fact that function Φ depends on k (whereas function Γ does not). In Figure 6 we also see that even though the three instances come from quite different families, the qualities obtained for $k > 10$ follow a remarkably similar pattern. We conjecture that the weight distribution bias in two of the instances and the sparsity of the third, combined with the superlinear growth of function Φ, make wrong edge choices very costly, producing this behavior.

It is worth noting the times required for $k = 1000$ and instance orlib625. Exact algorithm: 8124.45 seconds; Heuristic A: 0.86 seconds; and Heuristic B: 2.85 seconds. These were the largest speedups of the heuristics with respect to the exact algorithm observed in this study.

Finally, it is also worth mentioning that the observed performance of the random algorithm with function Φ was similar to that with function Γ. The best results were obtained for instance orlib625, where quality values were between 22% and 37% of optimal for $2 \leq k \leq 10$. This is to be expected, since instance orlib625 is so sparse, and hence edge choices are few. For the other instances quality values were much worse. For example, on instance hyperb256 and $k = 100$ quality was 36.8.

4.2.3 *Comparison of heuristics A and B.* The last experiment compares the performance of heuristics A and B on family random, $n = 500$, $k = 1000$, function Γ, and edge weights between 1 and m. We varied the number of edges from $m = 1200$ (very sparse) to $m = 124750$ (complete), on a total of 16 m values (25 instances for each value of m). We present three graphs: Figure 7 compares both heuristics in terms of *find* operations (on the *union-find* data structures used). Figure 8 com-

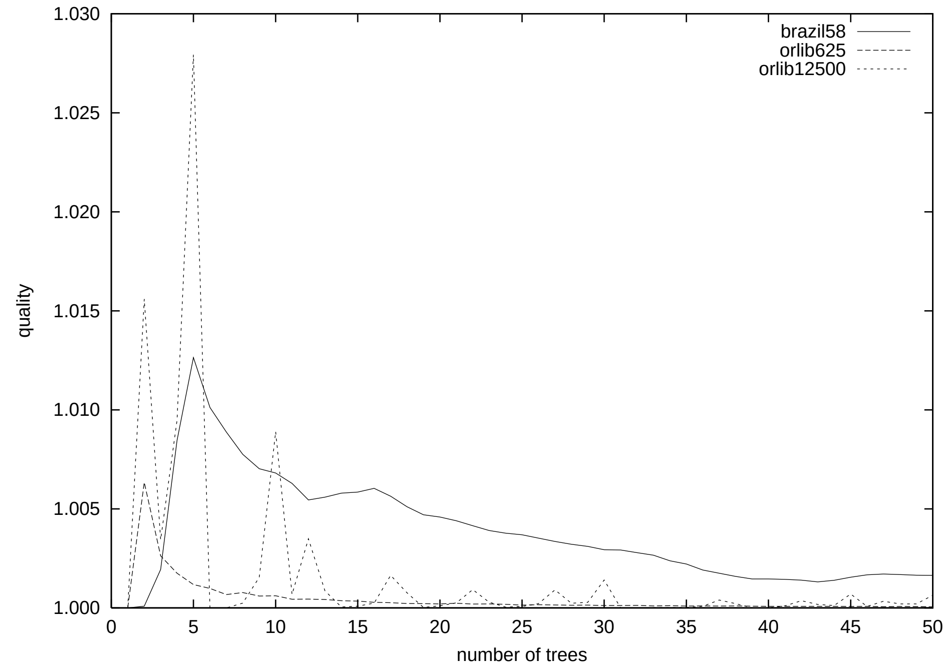

Fig. 3. Quality values for small values of k, second batch, penalty function Γ.

pares both heuristics in terms of *edge comparison operations*. An edge comparison operation is a comparison between incremental costs of two edges. These two kinds of operations encapsulate the essential basic operations of these two heuristics. Finally Figure 9 compares both heuristics in terms of running time (with operation counts included in the code).

As can be seen in the figures, the two heuristics have very different behaviors with respect to these instances. Heuristic A has a km factor on its running time, caused by the merging step alluded to in its description (see Section 3.1), and that explains Figure 8 and Figure 9. The explanation for the drop in number of *find* operations for $m > 30000$ seems to be that it becomes easier to find valid edges (and hence complete trees) when more edges are available. Note also that variance in operation counts (and time) for Heuristic A is high. Standard deviations observed were in some cases as high as 50% of the mean.

Heuristic B has almost constant performance with respect to increasing m on all three measures. The reason is that the running time is dominated by the edge insertion operation. Each such operation takes on average $O(\alpha(2n, n) \log n + \log m)$ time. While the dependence on m exists, it is very weak.

The conclusion from this last experiment is that Heuristic B is faster for dense graphs, while A may be faster for sparse graphs. This is consistent with results seen in Tables 3, 4, 5, and 6.

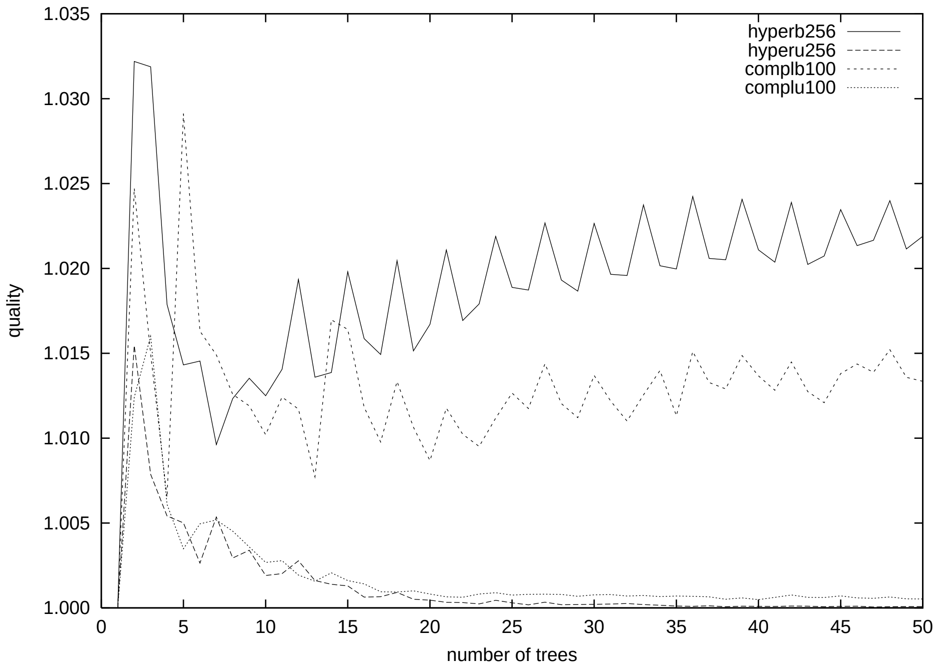

Fig. 4. Quality values for small values of k, first batch, penalty function Φ.

5. FINAL REMARKS

The parallel edge technique is a standard way of dealing with certain nonlinear optimization problems, such as the kMSTc problem, studied here. We have shown that greedy heuristics, well implemented, represent an interesting alternative to solve this problem, if optimality is not crucial but running time is. In addition these heuristics obtain results that are far better than random solutions.

We have also investigated the possibility of adding a lower bound computation to our heuristics. A simple lower bound for the kMSTc problem is a set of $k(n-1)$ edges (not necessarily distinct) of G such that the sum of their penalized weights is minimum and such that no more than k copies of any given edge are selected. The computation of this lower bound can be done as follows. Initially, we must insert every edge $e \in E$ into a priority queue and set $counter_e = 0$. In each step of the algorithm, we remove the first edge (e) from the queue and increment $counter_e$. If the new value of $counter_e$ is k, e can be discarded; if $counter_e < k$, we must update the incremental cost of e and reinsert it into the queue. We stop after $k(n-1)$ steps. If the priority queue is implemented as a binary heap, each step will require $O(\log m)$ time. Adding the $O(m)$ time necessary to build the heap, this yields a running time of $O(m + kn \log m)$, asymptotically better than any of the algorithms discussed.

This lower bound is so simple that its computation can be incorporated into any of the heuristics, thereby creating a program that will output a quality measure of the solution obtained, while still being much faster than the exact algorithm. However,

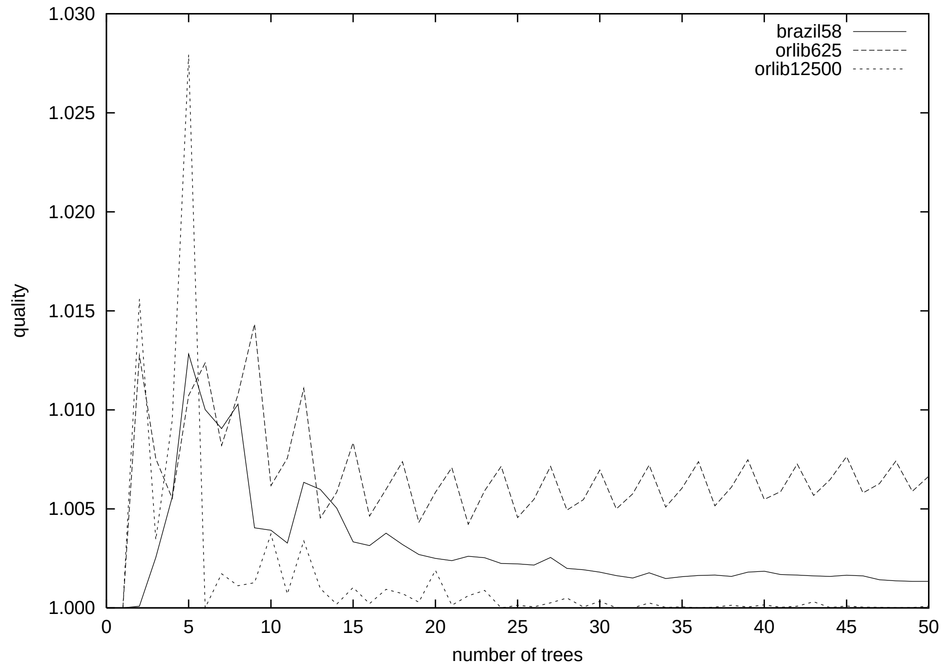

Fig. 5. Quality values for small values of k, second batch, penalty function Φ.

preliminary experiments have shown that this lower bound is not consistently strong for this purpose (there are cases in which it attains only 50% of the optimum value; in many other cases, however, it did reach 99% of optimal). We are currently trying to improve this bound and at the same time keep it fast to compute.

Several other lines of future research suggest themselves: is there a faster exact algorithm for kMSTc? What is the behavior of the heuristics for other penalty functions? Using the penalty functions presented here, are the heuristics in fact approximation algorithms? In what other congestion problems can these greedy techniques be applied? The kMSTc problem without the weak convexity condition (relaxing it by requiring for example that w_p be just nondecreasing with respect to i_e) is also worth investigating.

ACKNOWLEDGMENTS

We thank J. Stolfi for fruitful conversations, A. Conceição for collaboration at an early stage, and the anonymous referees for comments that helped improve the paper. In particular, we owe Section 3.3 to a referee's insight.

REFERENCES

Ahuja, R., Magnanti, T., and Orlin, J. 1993. *Network flows: Theory, Algorithms, and Applications*. Prentice-Hall.

Beasley, J. E. 1990. OR-library: Distributing test problems by electronic mail. *Journal of the Operations Research Society 41*, 1069–1072.

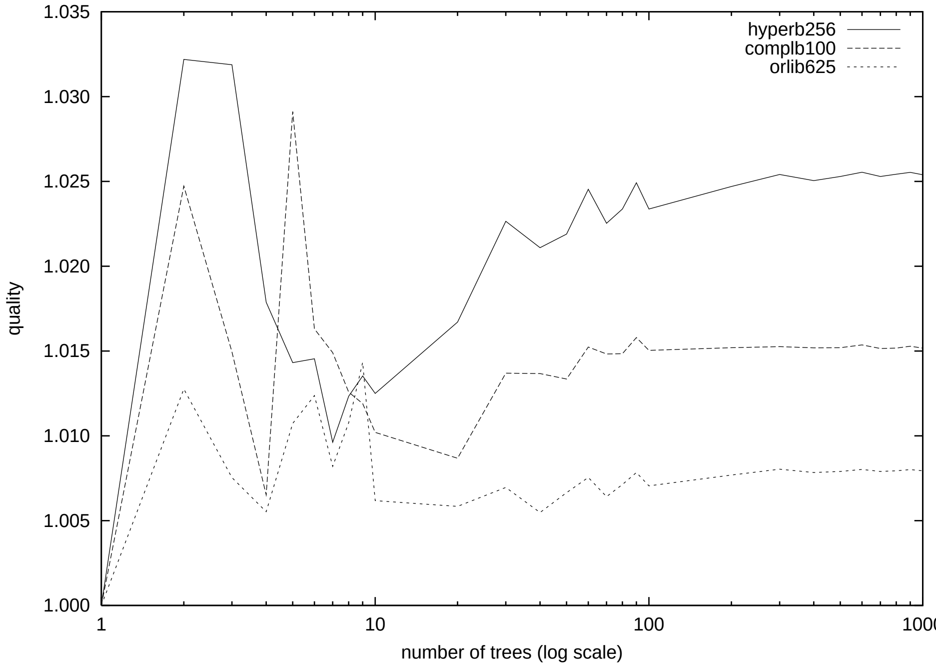

Fig. 6. Quality values for hard instances and large k, penalty function Φ.

Cormen, T., Leiserson, C., and Rivest, R. 1990. *Introduction to Algorithms*. MIT Press/McGraw Hill.

Edmonds, J. 1965a. Lehman's switching game and a theorem of Nash-Williams. *Journal of Research of the National Bureau of Standards 65B*, 73–77.

Edmonds, J. 1965b. Minimum partition of a matroid into independent subsets. *Journal of Research of the National Bureau of Standards 65B*, 67–72.

Fortz, B. and Thorup, M. 2000. Internet traffic engineering by optimizing OSPF weights. In *Proceedings of IEEE INFOCOM* (March 2000). IEEE.

Leighton, T., Makedon, F., Plotkin, S., Stein, C., Tardos, É., and Tragoudas, S. 1995. Fast approximation algorithms for multicommodity flow problems. *Journal of Computer and System Sciences 50*, 2, 228–243.

Nash-Williams, C. 1961. Edge-disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society 36*, 445–450.

Roskind, J. and Tarjan, R. 1985. A note on finding minimum-cost edge-disjoint spanning trees. *Mathematics of Operations Research 10*, 4, 701–708.

Tarjan, R. E. 1975. Efficiency of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery 2*, 212–225.

Tutte, W. 1961. On the problem of decomposing a graph into n connected factors. *Journal of the London Mathematical Society 36*, 221–230.

Williamson, D. P. and Goemans, M. X. 1996. Computational experience with an approximation algorithm on large-scale Euclidean matching instances. *INFORMS Journal on Computing 8*, 29–40.

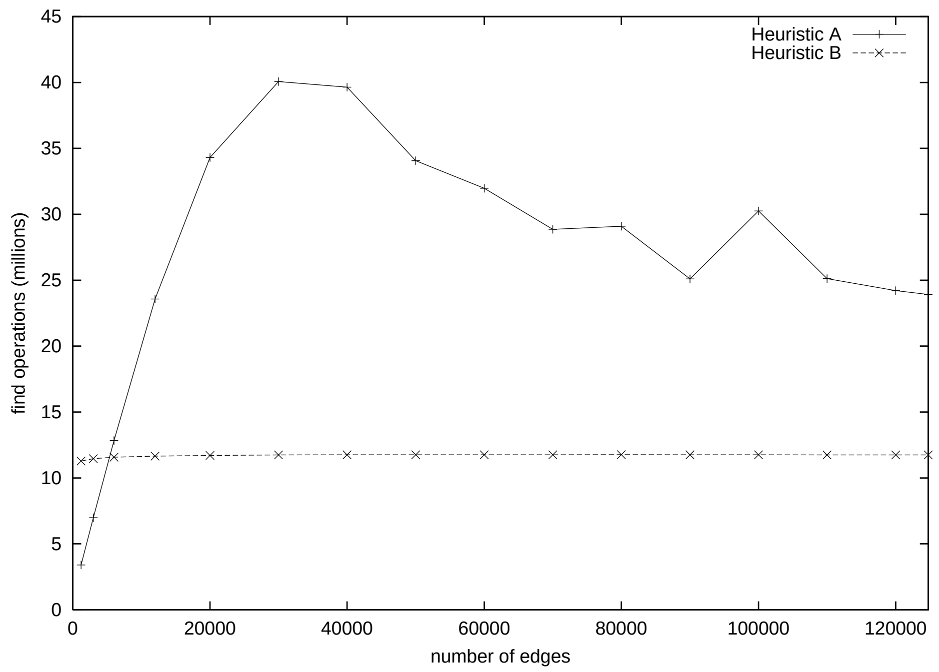

Fig. 7. Comparison of Heuristics A and B on instances from family random ($n = 500$ and $k = 1000$) in terms of *find* operations.

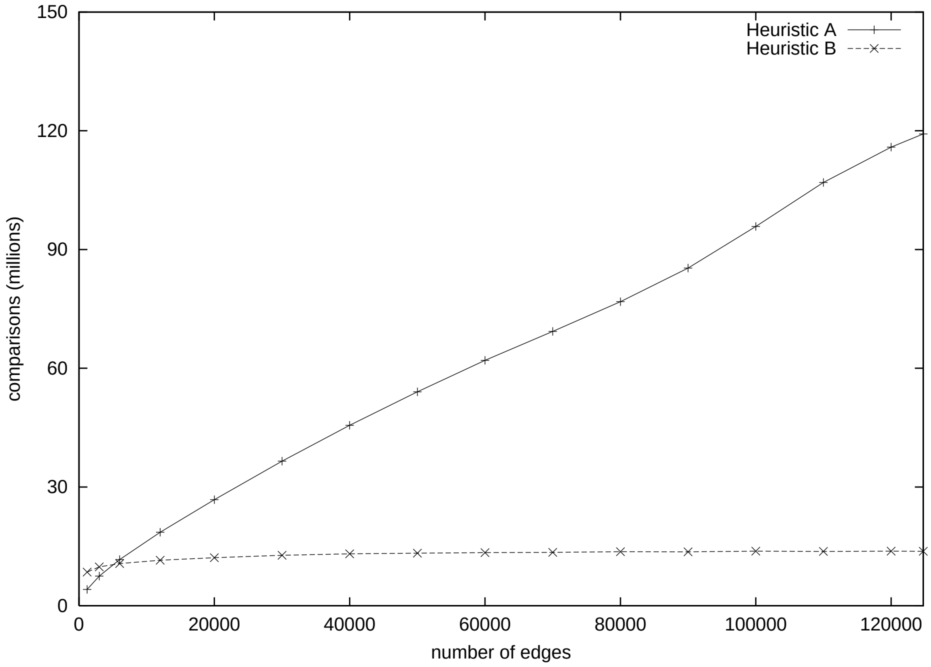

Fig. 8. Comparison of Heuristics A and B on instances from family random ($n = 500$ and $k = 1000$) in terms of edge comparison operations.

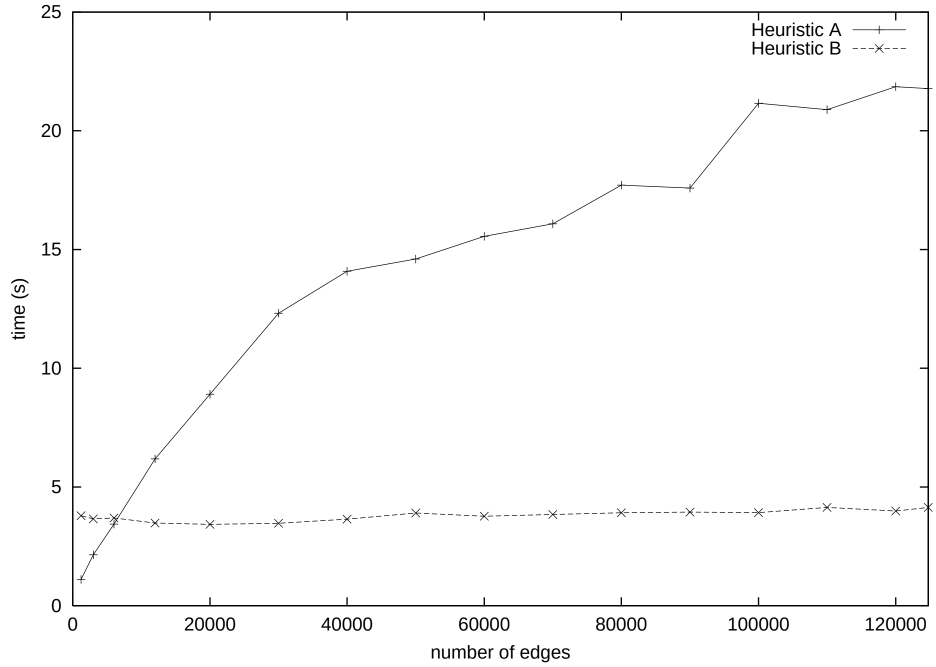

Fig. 9. Comparison of Heuristics A and B on instances from family random ($n = 500$ and $k = 1000$) in terms of running time.