A Distributed Dual Ascent Algorithm for Steiner Problems in Multicast Routing

Lúcia M.A. Drummond, Marcelo Santos
({lucia,mpinto}@ic.uff.br)
Department of Computer Science
Eduardo Uchoa (uchoa@producao.uff.br)
Department of Production Engineering
Fluminense Federal University, Brazil

Abstract

Multicast routing problems are often modeled as Steiner Problems in undirected or directed graphs, the latter case being particularly suitable to cases where most of the traffic has a single source. Sequential Steiner heuristics are not convenient in that context, since one cannot assume that a central node has complete information about the topology and the state of a large wide area network. This paper introduces a distributed version of a Dual Ascent primal-dual heuristic, known for its remarkably good practical results, lower and upper bounds, in both undirected and directed Steiner problems. Complexity analysis and experimental results are also presented, showing the efficiency of the proposed algorithm when compared with the best distributed algorithms in the literature.

keywords: Distributed Algorithms, Steiner Problem in Networks, Multicast Routing.

1 Introduction

The Steiner Problem in Graphs (SPG) is defined as follows. Given an undirected graph $G = (V, E)$, positive edge costs c and a set $T \subseteq V$ of *terminal nodes*, find a connected subgraph (V', E') of G with $T \subseteq V'$ minimizing $\sum_{e \in E'} c_e$. In other words, find a minimum cost tree containing all terminals, possibly also containing some non-terminal nodes. The Steiner Problem in Directed Graphs (SPDG) considers set T in a directed graph $G_D = (V, A)$, with a so-called root $r \in T$. The problem is to find a minimum cost directed tree containing paths from r to every other terminal. Both the SPG and the SPDG are NP-hard, one must resort to heuristic algorithms when solutions are to be obtained quickly.

Several emerging network applications, like teleconferencing or video on demand, require the transmission of large amounts of data among a small subset of the nodes. This is called *multicast* or *selective broadcast*, the usual broadcast being the case where the information must be sent to all nodes in the network. The routing of multicast connections is the problem of establishing message paths for a multicast session. Such routing problems are often modeled as a SPG, see surveys of Novak et al. [10] and Oliveira and Pardalos [12]. The frequent situation where most multicast messages have a single source and the network is asymmetric, i.e., link characteristics like latency, capacity, congestion or price depend on the direction, is better modeled as a SPDG. Centralized Steiner heuristics are not very suitable for multicast routing, normally one cannot

assume that a central node has complete information about the topology and the state of a large wide area network. The overhead to collect, store and update this information could be prohibitive. In this context, there is a need for distributed algorithms, where in each node there is a limited knowledge, only about the immediate neighborhood.

The first proposed approach to this distributed problem [3] was constructing a minimum cost spanning tree, using the algorithm described in [4], and removing subtrees without terminals, until a Steiner tree is obtained. However, this simple heuristic usually leads to poor solutions. Shaik [18] introduced DDMC (Destination Driven Multicast), an adaptation of the shortest path tree algorithm that favors paths passing through terminal nodes. The solutions obtained by DDMC are still not good. More sophisticated algorithms, distributed versions of sequential heuristics specially devised for Steiner problems and already known to yield good solutions (like those surveyed in [20]), had been proposed in the last 10 years.

The Prim Shortest Path Heuristic (Prim-SPH) grows a single tree, starting with a chosen terminal, called the root. At each step a least cost path is added, from the existing partially built tree to a terminal not yet connected. Bauer and Varma [1] and Regelj and Klavzar [16] proposed distributed versions with a time complexity of $O(|T|.|V|)$ (measured by the maximum sequence of messages) and $O(|V|^2)$ exchanged messages. Novak et al. [11] proposed improvements on those algorithms leading to a better practical performance, but could not change those worst case complexities. The above mentioned distributed versions of Prim-SPH can be used on both SPG and SPDG.

The so called Kruskal-SPH (although it actually resembles Borüvka's minimum spanning tree algorithm) grows several subtrees at once, starting at each terminal. At each step some pairs of subtrees are joined by least cost paths. Its distributed version was proposed by Bauer and Varma [1], with complexities $O(|T|.|V|)$ time and $O(|V|^2)$ messages. This last complexity was improved to $O(|V| \log |V|)$ by Singh and Vellanki [19]; this also improves the time complexity when $|T|$ is not $O(\log |V|)$. Those algorithms (and even the sequential Kruskal-SPH) were designed for the SPG and cannot be adapted to the SPDG.

The Average Distance Heuristic (ADH) also starts with subtrees composed by each terminal. At each step a pair of subtrees is joined by a path passing by the node (terminal or not) with minimum average distance to each subtree. The distributed version by Gatani et al. [5] takes $O(|T|.|V|)$ time and $O(|E| + |T|.|V|)$ messages. The ADH cannot be used on the SPDG.

It is important to remark that all those distributed algorithms (Prim-SPH, Kruskal-SPH and ADH) assume that each node already knows the values of least cost paths to all other nodes. If this is not the case, the distributed computation of such values adds a message complexity of $O(|E|.|V|)$ and time complexity of $O(|V|)$ [17].

The SPG and SPDG heuristic proposed in this article is a distributed version of the Dual Ascent (DA) algorithm proposed by Wong [22], followed by the application of the distributed Prim-SPH over the small restricted subgraph provided by DA. This complete Dual Ascent Heuristic (DAH = DA + Prim-SPH) has the following advantages over other distributed heuristics.

- Extensive computational experiments over the main classes of SPG benchmark instances from the literature have shown that DAH usually yields better solutions [13, 21, 15, 14] than Prim-SPH. Solutions by Kruskal-SPH and ADH are usually a little worse than those by Prim-SPH.

- DA can be interpreted as working in the dual of a linear program formulation. In practice, this means that DAH not only returns a solution, it also returns a lower bound that gives

a guarantee of the quality of this solution. DA lower bounds are remarkably good, they are usually less than 3% below the optimal.

- The distributed DA does not require that nodes know the value of least cost paths to every other node. Some authors [1, 11, 5] argue that network layer protocols like RIP and OSPF already keep routing tables with that information, they can be used by their Steiner algorithms. Such reasoning is not completely satisfactory since it relies on particular technologies that may be supplanted in the future. Moreover, using network layer information limits what can be accomplished by the multicast protocol. RIP routing tables actually provides hop distances, i.e., least cost paths assuming that all links have unitary costs. OSPF routing tables use the link costs provided by local administrators (usually as a function of parameters like latency time or available bandwidth). RIP or OSPF costs are not necessarily the more appropriate for building multicast trees. This application may prefer using its own link costs, reflecting factors like contractual prices or forecasted link behavior after the multicast begins. Note that the Prim-SPH performed in DAH after DA does require the computation of least cost paths. However, this happens in a small restricted graph.

The proposed distributed DA has worst case complexities of $O(|T|.|V|^2)$ messages, $O(|T|.|V|^2)$ global time, and $O(|V|)$ local time. An alternative to perform the DA would be electing a leader node to locally solve the problem and then broadcast the solution. It would take $O(|V|.|E|)$ messages and $O(|V|)$ time to concentrate all the relevant information about G in the leader, the local time complexity of the sequential DA algorithm is $O(|E|.\min\{|E|, |T|.|V|\})$ [8]. Considering such complexities and the memory demand at the leader node, the fully distributed approach proposed is an appealing alternative for multicast routing.

The remainder of this paper is organized as follows. Section 2 introduces the sequential version of Wong's Dual Ascent Algorithm. The distributed DA algorithm is presented and analyzed in Section 3. Section 4 presents experimental results, comparing the practical performance of DAH with Prim-SPH and DDMC in terms of solution quality, exchanged messages and time. Those experiments are relevant because the empirical performance of the distributed DA is much better than its worst case complexities. The last section contains conclusions.

2 Sequential Version of Wong's Dual Ascent Algorithm

Wong's Dual Ascent (DA) is an algorithm for the SPDG. However, any SPG can be turned into a SPDG, allowing the effective use of DA in this case too. The transformation defines $G_D = (V, A)$ as the directed graph obtained by replacing each edge in E by two opposite arcs and chooses any terminal r to be the root. Each pair of opposite arcs receives the same cost of the original edge. One now seeks in G_D for a minimum cost arborescence (a directed tree) having paths from r to every other terminal. Such arborescence corresponds to an optimal solution of the SPG in the original graph.

In this algorithm, each arc a has its non-negative *reduced cost* $\bar{c}_a$, a value that is initialized with the original arc cost. Reduced costs may be only decreased. Arcs with zero reduced cost are called *saturated*. Those arcs induce the *saturation graph* $G_S = (V, A_r(\bar{c}))$, where $A_r(\bar{c}) = \{a \in A : \bar{c}_a = 0\}$. As all original costs are positive, this graph starts with no arcs. All DA operations are defined over the current saturation graph. Let R be the set of nodes of a strongly connected component of G_S, i.e., a maximal subgraph containing a directed cycle going through

any pair of nodes in it. This set is a *root component* if (i) R contains a terminal, (ii) R does not contain r, and (iii) there is no terminal $t \notin R$ reaching R by a path in G_S. Given a root component R, define $W(R) \supseteq R$ as the set of nodes reaching R by paths in G_S and let $\delta^-(W)$ be the directed cut consisting of the arcs entering W. The DA algorithm follows:

Wong's Dual Ascent
$LB \leftarrow 0$;
$\bar{c}_a \leftarrow c_a$, for all $a \in A$;
While (root components exist in G_S) {
    Choose a root component R;
    $W \leftarrow W(R)$;
    $\Delta \leftarrow \min_{a \in \delta^-(W)} \bar{c}_a$;
    $\bar{c}_a \leftarrow \bar{c}_a - \Delta$, for all $a \in \delta^-(W)$;
    $LB \leftarrow LB + \Delta$;
}
Output: LB and $\bar{c}$

At first, each terminal other than the root corresponds to a root component. In each round, a root component R is chosen and the reduced costs of all arcs entering $W(R)$ are decreased by Δ, the smallest such reduced cost. The partial lower bound is increased by the same amount. At least one arc is saturated in each round. Some saturations reduce the number of root components, until there are no root components anymore. At this point, G_S contains at least one directed path from r to every other terminal.

As output DA returns a lower bound as well as the final reduced costs. In order to obtain a feasible SPDG solution, Wong proposed using an additional heuristic over G_S:

1. Let Q be the set of nodes that can be reached from the root. Compute a minimum cost spanning arborescence on the subgraph of G_S induced by Q.

2. Remove from this arborescence all leaves that are not terminals. Repeat this pruning step until all leaves are terminals.

Other authors [15, 13] found that running Prim-SPH over G_S is very fast (because this graph is usually very sparse) and yields solutions that are even better than those provided by the method proposed by Wong. Anyway, the final output obtained after "DA + Prim-SPH over G_S" (we call this DAH), is a Steiner tree; further the lower bound LB gives a guarantee of the quality of this solution. It should be noted that, although very tight guarantees are obtained in practice, this is not an approximative algorithm. In fact, no algorithm approximating the SPDG by a constant factor is possible unless $P = NP$ [6].

3 An Asynchronous Distributed Version of DA

3.1 Assumptions and Terminology

We assume that each node knows the cost of each arc entering that node. During the execution, all data about the state of an arc is kept by its adjacent nodes. Each node performs the same local algorithm, which consists of sending messages over adjoining links, waiting for incoming

messages, and processing. Messages can be transmitted independently and arrive after an unpredictable but finite delay, without error and in sequence. We view the nodes in the graph as being initially asleep. All non-root terminals wake up spontaneously, other nodes awake upon receiving messages from awakened neighbors. We assume that the directed graph is bidirectional, i.e., if arc (i, j) exists then (j, i) also exists (or must be added, with infinity cost). We also assume that the nodes of the graph have distinct identities that can be totally ordered.

The agents in this distributed algorithm are associated with the *fragments*. A fragment is defined as a set $W(t)$ formed by all nodes reaching a non-root terminal t by a path of saturated arcs. The terminal t identifying a fragment $W(t)$ is also known as its *leader*. Let R be the strongly connected component containing t, $W(t) = W(R)$. A node can operate on behalf of several fragments, because it can belong to different fragments.

By definition, all nodes in a fragment are connected to its leader by saturated arcs. Among such arcs, the algorithm keeps in each fragment a tree used for intra-fragment message exchange, *convergecast messages* from nodes in the fragment to the leader and *broadcast messages* from the leader to the fragment's nodes. A fragment *growing round* consists of finding the minimum reduced cost of an entering arc and subtracting this value from the reduced cost of all entering arcs. The first operation is performed using a convergecast, the result is then broadcast. When decreasing the reduced costs of fragment-entering arcs, some new arcs become saturated, the corresponding nodes must be included into the fragment and the fragment's tree is updated. The fragment leader keeps the partial lower bound due to the fragment growing rounds. As fragments grow, their nodes may start to overlap. Having common entering arcs creates a mutual exclusion problem, that causes some fragments to suspend their operations temporarily. A fragment finishes its execution when it is reached from the root. When the root reaches all terminals, it initiates a broadcast to terminate the algorithm.

3.2 Growing Fragments

At first, a fragment composed by just a terminal t selects the entering arc with the minimum reduced cost, at this point the original cost. The fragment partial lower bound is increased from zero to this value, which is subtracted from the reduced cost of each entering arc. Messages *Include(t)* are transmitted on the saturated arcs. The node sending the *Include(t)* messages, marks those arcs as *To_leaf(t)*. Those arcs define a directed tree from the leader to all other nodes in the fragment, for broadcast operations. Upon receiving this message, a node includes itself into the fragment and marks the outgoing arc to the node that sent the message as *To_leader(t)*. The arcs marked as *To_leader(t)* define a directed tree pointing to the leader, used in convergecast operations. This node also sends *Check(t)* messages to all its other neighbors, communicating that it now belongs to t and asking if they are also part of t. Those messages must be answered with an *Ack(t, true)* message if the neighbor also belongs to t and *Ack(t, false)* if it does not.

The next step is calculating the minimum reduced cost of all arcs entering the fragment. A leaf node in the fragment sends on its *To_leader(t)* arc a message *Conv(t,Smallest,δ)* with the minimum reduced cost of an arc entering it, δ, and not belonging to the fragment. The internal node in the fragment, upon receiving its *Conv(t,Smallest,δ)* messages, compares those costs with the minimum reduced cost of an arc entering it and not belonging to the fragment. The smallest such value is sent on its *To_leader(t)* arc using another *Conv(t,Smallest,δ)* message. When the fragment leader finally knows the fragment minimum reduced cost, called Δ, it increases the fragment's lower bound and starts broadcasting Δ back to all fragment nodes, using *Broad(t,Smallest,Δ)* messages. Upon receiving such a message, a node decreases the cost

of its entering arcs, which may trigger *Include(t)* messages on the newly saturated arcs, therefore growing the fragment. A newly added node first checks its neighborhood before starting another round of convergecast. Leaf nodes that did not include any new node start a new round of convergecast immediately. See Figure 1, for an example.

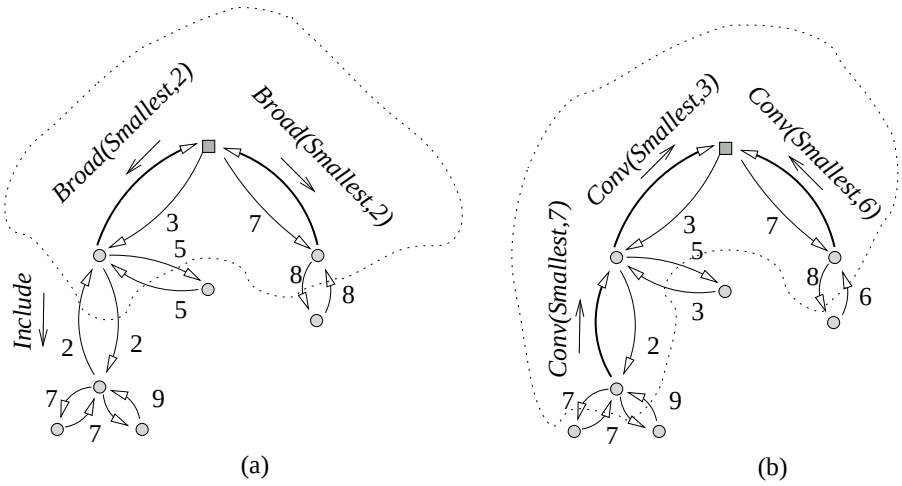

Figure 1: Growing Fragments

When several nodes are included into a fragment in the same growing round, it is possible that a newly added node i receives an *Ack(t,false)* from a node j that will still receive an *Include(t)* in that round. This means that i will consider arc (j, i) as a fragment-entering arc and possibly its reduced cost reaches the leader as the smallest. When a *Broad(t,Smallest,*Δ*)* with this value reaches i, this node will have already been informed (by a *Check(t)* message) that j is actually part of the fragment. The reduced cost of arc (j, i) is not decreased. Therefore, it is possible to have *degenerated growing rounds*, i.e., rounds where no arc is saturated and no node is included. However, every degenerated growing round will be followed by a non-degenerated one, because the subsequent round cannot again consider fragment-internal arcs as fragment-entering.

Another situation happens when nodes i and j send *Include(t)* messages to the same node k in the same growing round. Suppose that the message from i arrives first. The second *Include(t)* message should be answered by an *AlreadyIncluded(t)* message. Node j then knows that (j, k) should not be marked as *To_leaf(t)*.

Finally, it is also possible that a newly included node also includes other nodes in the same growing round. Suppose that node i is included. It sends *Check(t)* messages and waits for the corresponding *Ack(t)* messages. Then it gets the minimum reduced cost of an entering arc not belonging to t. If this value is positive, i knows that it is a fragment leaf and sends a *Conv(t,Smallest,*δ*)* message as usual. However, since other fragments are also decreasing reduced costs, it is possible that this value is zero, i.e., there are saturated arcs (j, i) such that j does not belongs to t. In this case, i sends *Include(t)* messages to nodes j, that will continue the growing round.

3.3 Suspending Fragments

Fragments with at least one common entering arc can not grow at the same time, since the reduced cost of these arcs would be changed concurrently. This situation represents a mutual

exclusion problem where the shared resource is the reduced cost of the common arc. In terms of the fragment growing framework of the previous section, this mutual exclusion problem arises when two or more fragments have a common node. In order to resolve this conflict, only the fragment with the largest identification number will continue growing, while the other fragments are suspended. They remain suspended until the fragment in conflict finishes its execution (growing). That is, at a given moment, we have the rule that each node is associated with at most one active fragment.

The above rule gives rise to two cases. When a node receives *Include(t_1)* from a fragment with an identification number smaller than its current active fragment t_2, it answers *Conv(t_1,Suspend)* immediately (like in the example in Figure 2, parts (a) and (b)). In the second case, where the identification number of t_1 is larger than t_2, the node must wait until t_2 finishes its current growing round, before suspending t_2 and continuing the growing of t_1. When a fragment t is suspended, the leader of the suspended fragment sends *Broad(t,Suspend)* messages to all its nodes.

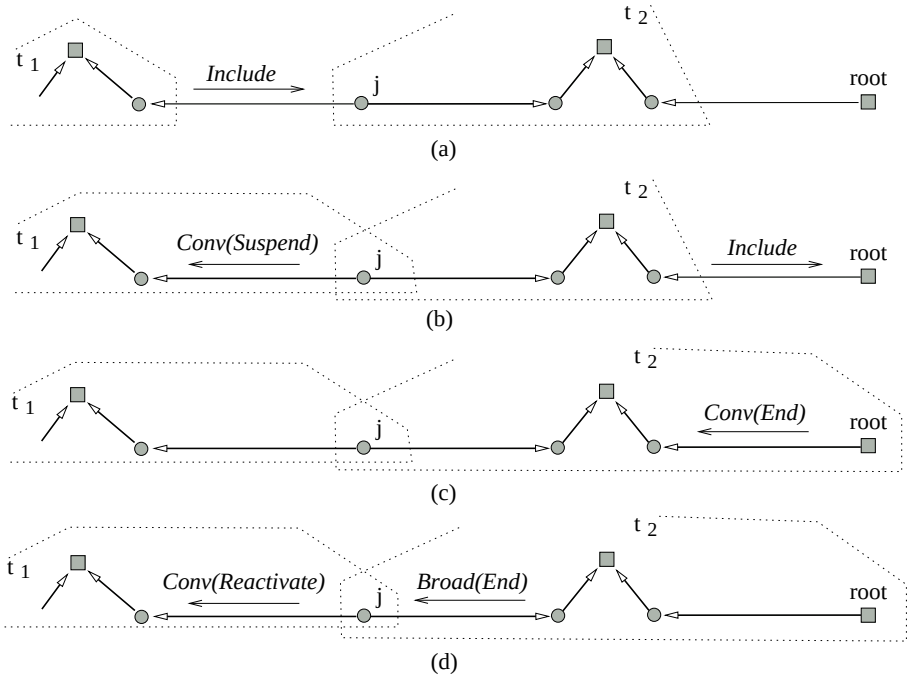

Figure 2: Suspending and Reactivating Fragments

3.4 Finishing and Reactivating Fragments

A fragment stops growing definitively when it is reached from the root node by saturated arcs. When the root receives an *Include(t)*, it sends back a *Conv(t,End)*. The fragment leader then sends a *Broad(t,End,pb)* with its partial lower bound and finishes.

When a node receives such a message indicating that its current active fragment was finished, it tries to reactivate the fragment with the largest identification number that was suspended by it, t'. This action is accomplished by sending a *Conv(t',Reactivate)* message (in the example of Figure 2, after t_1 is suspended, t_2 finishes, then node j tries to reactivate t_1). Then the leader of t' sends *Broad(t',Reactivate)* messages. Note that a reactivation attempt may fail because t' still

has other conflicts at a node k other than j with non-finished fragments with larger identification number. In this case, k answers *Broad(t',Reactivate)* with another *Conv(t',Suspend)* message.

The root accumulates partial lower bound values to obtain the global lower bound. When the root reaches all terminals, it initiates a broadcast to terminate the algorithm.

3.5 Analysis of the Algorithm

3.5.1 Correctness

Theorem 1 *The algorithm guarantees that the root reaches all terminals in a finite time.*

Proof: By the description of Subsection 3.2, for each two consecutive growing rounds, at least one node is included into a fragment. An active fragment makes growing rounds until it is suspended or finished. By the description in Subsection 3.3, it follows that mutual exclusion is guaranteed in the access to entering arcs of a shared node and that in such a conflict only the active fragment with the largest identification number will not be suspended. Therefore, at least the current non-finished fragment with the largest identification number, t_j, is always making growing rounds and is reached by the root after at most $2.|V|$ such rounds. Now, let t_i be the new non-finished fragment with the largest identification number. Either t_i was already active or it was suspended by t_j. In the last case, t_i is reactivated by t_j. ■

3.5.2 Communication Cost

We want an upper bound on the number of messages exchanged during the execution of the algorithm. When a node is included into a fragment, it checks at most $|V|$ neighbors, this is done at most $|V| - |T|$ times for each of the $|T| - 1$ fragments. Therefore, there can be up to $O(|T|.|V|^2)$ *Check* and *Ack* messages. We now show that no other message exceeds that bound. Each growing round requires at most $O(|V|)$ *Conv*, *Broad* and *Include* messages. As there can be up to $O(|V|)$ growing rounds for each fragment, $O(|T|.|V|^2)$ is a valid upper bound on the number of such messages.

Considering the suspending and reactivation procedures, a reactivation is only triggered when a fragment is finished. So each fragment can be reactivated only once by each other fragment. The total of reactivations is $O(|T|^2)$. For each reactivation, a broadcast on the tree is executed, that never requires more than $O(|V|)$ messages. So, $O(|T|^2.|V|)$ is a valid upper bound.

That upper bound is asymptotically tight. A worst case occurs, for example, in a complete graph where all non-terminal nodes are included into all fragments. In this case, there are $\Theta(|T|.|V|^2)$ *Check* and *Ack* messages.

3.5.3 Global Time Cost

The global time complexity in the asynchronous model adopted assumes that local computations take no time, and also that the time to communicate one message to each node of a nodes set of neighbors is O(1). The complexity is then the number of messages in the longest causal chain of the form "receive a message and send a message as a consequence" occurring in all executions of the algorithm and over all applicable variations in the structure of the graph [2]. This time complexity is never larger than the message complexity. We next show that there exists a case where DA requires $\Theta(|T|.|V|^2)$ time.

The worst time situation in DA is when only one fragment can grow at a time, because all other non-terminated fragments are suspended. The growing of a single fragment may take $O(|V^2|)$ time. Each growing round requires a causal chain of *Broad* and *Conv* messages that is proportional to the longest path of saturated arcs in the current broadcast/convergecast tree of the fragment. This longest path may have $O(|V|)$ length and the complete growing of this fragment may take $O(|V|)$ growing rounds. Even in this situation, the total time to grow all $|T|$ fragments may still be quadratic. This happens because several nodes can be included into a fragment in a single growing round, since the algorithm takes advantage of arcs that were saturated in the growing of previous fragments. However, in the following case this is not enough to avoid a cubic complexity.

Define a complete directed graph with $m + 2n + 2$ nodes, $n \geq m$, composed by node sets $T = \{t_1, \ldots, t_{m=|T|}\}$, $U = \{u_1, \ldots, u_n\}$, $W = \{w_1, \ldots, w_n\}$, plus nodes r and s. Node r is the root, nodes in T are terminals and the remaining nodes are non-terminals. All fragments include s in the beginning of DA, causing a conflict among all fragments. The fragment t_m will continue its growing, the others will be suspended. Suppose this fragment makes n growing rounds, including nodes u_1 to u_n and forming a path of saturated arcs of length n. Fragment t_m then makes more n growing rounds, this time saturating the arcs from W to the terminal t_m, one by one. Each such growing round takes $\Theta(n)$ time. Finally, t_m includes the root, saturating the arc (r, t_m). The complete growing of this fragment takes $\Theta(n^2)$ time. Now, fragment t_{m-1} is reactivated and it includes node u_1. All nodes in U are included in this same growing round. This means that all those n nodes were included in $O(n)$ time, but now fragment t_{m-1} has a path in its broadcast/convergecast tree of length n. Fragment t_{m-1} makes more n growing rounds, saturating the arcs from W to the terminal t_{m-1}, one by one. Each such growing round takes $\Theta(n)$ time. This fragment finishes after including the root. Its complete growing also takes $\Theta(n^2)$ time. If all remaining fragments behave in a similar way, the complete DA will take $\Theta(m.n^2) = \Theta(|T|.|V|^2)$ global time.

3.5.4 Local Costs

The local time complexity refers to the time to perform local computations in a node between the receiving of two consecutive messages. The local space complexity is the maximum amount of memory that must be kept by a node during the execution of the algorithm. As can be checked in the pseudocode shown in Appendix A, the local time spent at a node i is $O(|T| + degree(i))$ and the local memory is $O(|T|.degree(i))$.

A distributed algorithm may be simulated in a single machine, charging $O(1)$ time for each message. The total sequential time took by this simulation is bounded by the message complexity times the local time complexity. In case of DA, the simulation would require $O(|T|.|V|^3)$ time in a complete graph. This is exactly the same complexity of the sequential DA.

4 Experimental Results

Our distributed algorithm was implemented in C and MPICH2-1.0.3 and executed on a cluster with 15 Athlon 1.8 GHz processors. Half of the tests was performed over the SPG benchmark instances from series B, I080 and P4Z series of Steinlib [9]. Instances from B series are random graphs with different densities and random edge costs between 1 and 10. Instances from I080 series are also random graphs with different densities, but edges costs were chosen to make them hard to solve. Instances from P4Z series have complete graphs. All instances in Steinlib are

undirected. Since the proposed algorithm was designed for the more general case of digraphs, we created SPDG instances from the above mentioned SPG instances. Each edge in the original graph is replaced by a pair of opposite arcs. Their costs are the original costs multiplied by a random factor uniformly distributed in the range [0.5,1.5] and rounded. A random terminal is chosen to be the root. We also implemented the distributed versions of the DDMC [18] and Prim-SPH [10] (Kruskal-SPH and ADH cannot be used in digraphs), including a shortest distance algorithm. The Prim-SPH was applied as a stand-alone algorithm and also in DAH to find a solution contained in the saturation graphs provided by the distributed DA algorithm.

Table 1 presents results on each individual directed instance, columns have the following meaning:

- $|V|$**,** $|E|$**, and** $|T|$ give the instance size;
- **Opt** is the value of the optimal solution (calculated with the branch-and-bound code described in [13]);
- **DDMC** is the cost of the solution obtained by DDMC algorithm;
- **SPH** is the cost of the solution obtained by Prim-SPH used as a stand-alone algorithm, over the whole graph G_D;
- **LB** is the lower bound given by DA;
- $\frac{|Ar|}{|A|}\%$ is the the proportion of the arcs that are saturated in the end of DA;
- **DAH** is the cost of the solution obtained by running Prim-SPH over the graph containing only those saturated arcs.

The results given for the DAH represent the average of 5 runs. Since processor load and communication times may change on each execution, the sequence of fragment growing in the DA may also change, leading to slightly different results. The average standard deviation of DAH solution cost in all executions is 1.5%.

We use *competitiveness*, the ratio of the heuristic cost and the optimal cost, to compare the quality of the solutions provided by each method. Charts in figures 3 and 4 show the cumulative percentage of cases whose competitiveness is less than or equal to a given value, for undirected and directed instances. It is clear that DAH gives better solutions than Prim-SPH and much better than DDMC. We also charted the improved results of executing both DAH and Prim-SPH and taking the best solution. However, there is a more interesting approach to obtain similar results. Using the lower bounds provided by DA, we can evaluate the solution obtained with DAH and execute Prim-SPH only if it exceeds a given limit. We applied this idea, executing Prim-SPH only if DAH / LB exceeded 1.06, this happened in 52% of the cases. Those results are also charted, in fact, the last two curves are undistinguishable.

We also compare the practical performance of DAH, Prim-SPH and DDMC performance with respect to time, given as the size of the largest message sequence, and to the total number of exchanged messages. Those measurements, for each directed instance, grouped by series, are shown in figures 5 and 6. We omit those measurements on undirected instances since the results are similar. It can be seen that DAH is indeed more costly than Prim-SPH on most instances, taking more time and exchanging more messages. However, it was never much more costly than Prim-SPH, their performance differences were always within a small factor of 4. Moreover, DAH performed better on larger I080 instances and much better on P4Z instances. The DDMC

Instance					DDMC	SPH	Dual Ascent		DAH
Name	$\|V\|$	$\|E\|$	$\|T\|$	Opt			LB	$\frac{\|Ar\|}{\|A\|}\%$	
b01d	50	126	9	77	107	82	77	13,9	**77**
b02d	50	126	13	101	145	107	100,2	13,5	**104**
b03d	50	126	25	170	**175**	176	166,2	9,8	175,4
b04d	50	200	9	58	75	61	56	13,4	**59,3**
b05d	50	200	13	61	85	**64**	61	13,3	65,1
b06d	50	200	25	128	179	134	127,2	13,1	**129,6**
b07d	75	188	13	122	201	**122**	121	13,7	125
b08d	75	188	19	115	172	126	114,8	13,8	**116,3**
b09d	75	188	38	240	282	244	239,2	13,5	**242,1**
b10d	75	300	13	90	119	**91**	89	13,7	92,6
b11d	75	300	19	103	143	**104**	100,8	13,3	105,2
b12d	75	300	38	168	269	**171**	161,8	11,6	173,7
b13d	100	250	17	165	**168**	202	161,7	13,4	169,4
b14d	100	250	25	235	**250**	261	235	13,4	258,8
b15d	100	250	50	342	370	356	327,2	13,8	**345**
b16d	100	400	17	133	170	**133**	133	13,9	141,6
b17d	100	400	25	139	180	**142**	137	13,5	145,2
b18d	100	400	50	258	266	271	258	13,8	**259,2**
i080-001d	80	240	6	1751	2276	1815	1750,5	15,7	**1788,3**
i080-101d	80	240	8	2322	3009	2706	2322	10,5	**2322**
i080-201d	80	240	16	4321	4335	4502	4228,4	22,6	**4328,8**
i080-301d	80	240	20	5714	7402	6365	5410,7	23,6	**5916**
i080-031d	80	320	6	1514	1970	1706	1451,8	16,2	**1552,8**
i080-131d	80	320	8	1875	2508	2061	1844	17,8	**2016**
i080-231d	80	320	16	4156	5403	4623	4083,2	14	**4410**
i080-331d	80	320	20	4506	5800	4911	4258,6	19,3	**4877**
i080-011d	80	700	6	1220	1585	1296	1220	17,4	**1227,6**
i080-111d	80	700	8	1580	2054	1600	1499,2	12,6	**1581,8**
i080-211d	80	700	16	2951	3825	3174	2862,8	12	**3145**
i080-311d	80	700	20	3471	4553	**3813**	3470,1	12,5	4082,3
i080-041d	80	1264	6	946	1230	1026	900	5,9	**983**
i080-141d	80	1264	8	1404	1825	1596	1265,6	10,5	**1404**
i080-241d	80	1264	16	2492	3250	2617	2357,7	9,8	**2546,4**
i080-341d	80	1264	20	3132	4072	**3361**	3007,1	15	3408,3
i080-021d	80	6320	6	741	961	847	652,9	5,8	**768,4**
i080-121d	80	6320	8	977	1250	1097	960,1	5,3	**1050**
i080-221d	80	6320	16	1985	2589	2293	1880,2	5,4	**2082,2**
i080-321d	80	6320	20	2453	3187	3137	2164,6	4,1	**2755,4**
P401d	100	9900	5	145	295	165	145	0,3	**145**
P402d	100	9900	5	102	152	**102**	102	0,2	**102**
P403d	100	9900	5	169	189	186	166	0,5	**180**
P404d	100	9900	10	270	305	**279**	199,7	0,3	288
P405d	100	9900	10	248	305	250	214,4	0,4	**248**
P406d	100	9900	10	281	350	303	226	0,3	**285,3**
P407d	100	9900	20	546	658	590	517,8	0,3	**572**
P408d	100	9900	20	502	821	520	490,6	0,3	**502**

Table 1: Solution quality on directed instances.

algorithm showed to be cheap in terms of messages, however the implementation proposed by its authors allowed almost no parallelism, leading to large times.

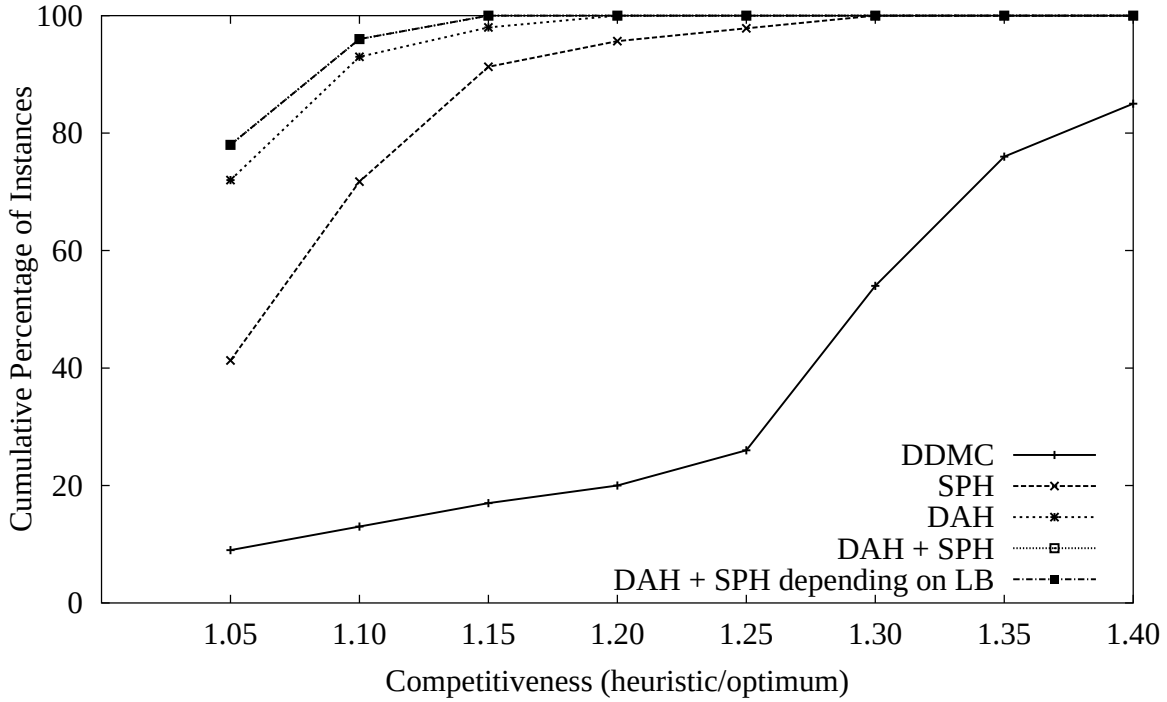

Figure 3: Solution quality – directed instances.

5 Conclusions

This work proposed a distributed version of Wong's DA algorithm for the Steiner problem in graphs that can be directly applied in a heuristic for multicast routing. The experimental evaluation of this algorithm, comparing its results with those from previous algorithms over a set of instances from the SteinLib, led to the following conclusions:

- *DAH indeed provides better solutions on average and also provides tight lower bounds giving strong guarantees on the quality of those solutions.* A similar conclusion was already established for the sequential counterpart of DAH [13, 21, 15]. However, those sequential algorithms are not identical to DAH, they may use some criteria to guide the fragment growing order (for example, grow the fragment leading to the minimum number of saturated arcs). Those global criteria are not implementable in a distributed algorithm. In fact, the fragment growing in the proposed distributed DA is not-deterministic and may be affected by message delays or processor load. The experiments showed that the lack of such a global criteria does not affect significantly the solution quality;

- *In spite of bad worst case complexities, the practical performance of DAH in terms of time and messages showed to be very competitive with those of previous algorithms.* This discrepancy between theory and practice was already noted in the sequential DA, that also

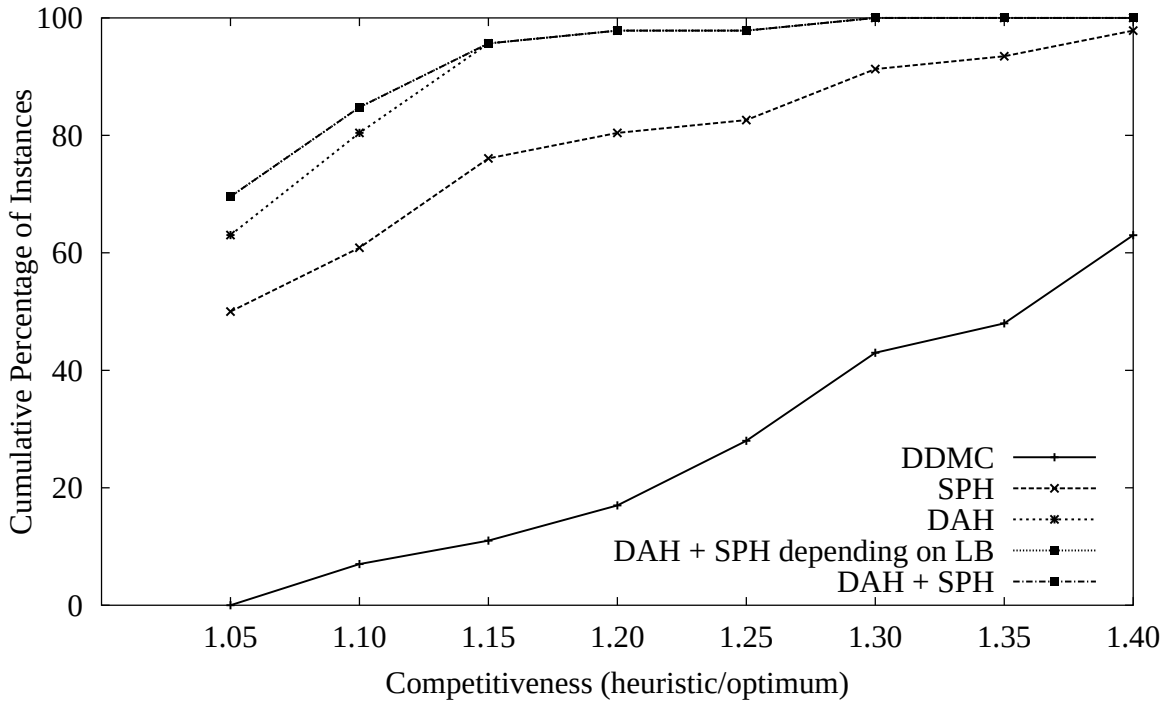

Figure 4: Solution quality – undirected instances.

has a bad time complexity but is usually fast [13, 21, 15]. The worst case complexities (in both sequential and distributed DA) are only observed in "pathological instances".

To summarize we conclude that DAH can be considered as a practical method for obtaining a provably good Steiner tree for multicast in a point-to-point network.

References

[1] F. Bauer and A. Varma, Distributed algorithms for multicast path setup in data networks, IEEE/ACM Transactions on Networking, 4 (1996) 181–191.

[2] V. Barbosa, An introduction to distributed algorithms, The MIT Press, (1996).

[3] G. Chen, M. Houle and M. Kuo, The Steiner problem in distributed computing systems, Information Sciences, 74 (1993) 73–96.

[4] R. Gallager, P. Humblet and P. Spira, A distributed algorithm for minimum-weight spanning trees, ACM Transactions on Programming Languages and Systems, 5 (1983) 66–77.

[5] L. Gatani, G. Lo Re and S. Gaglio, An efficient distributed algorithm for generating multicast distribution trees, Proc. of the 34th ICPP – Workshop on Performance Evaluation of Networks for Parallel, Cluster and Grid Computer Systems (2005) 477–484.

[6] M. Goemans and D. Williamson, The primal-dual method for approximation algorithms and its application to network design, In: Approximation Algorithms for NP-hard problems, D. Hochbaum (Ed.), PWS Publishing, (1997) 144–191.

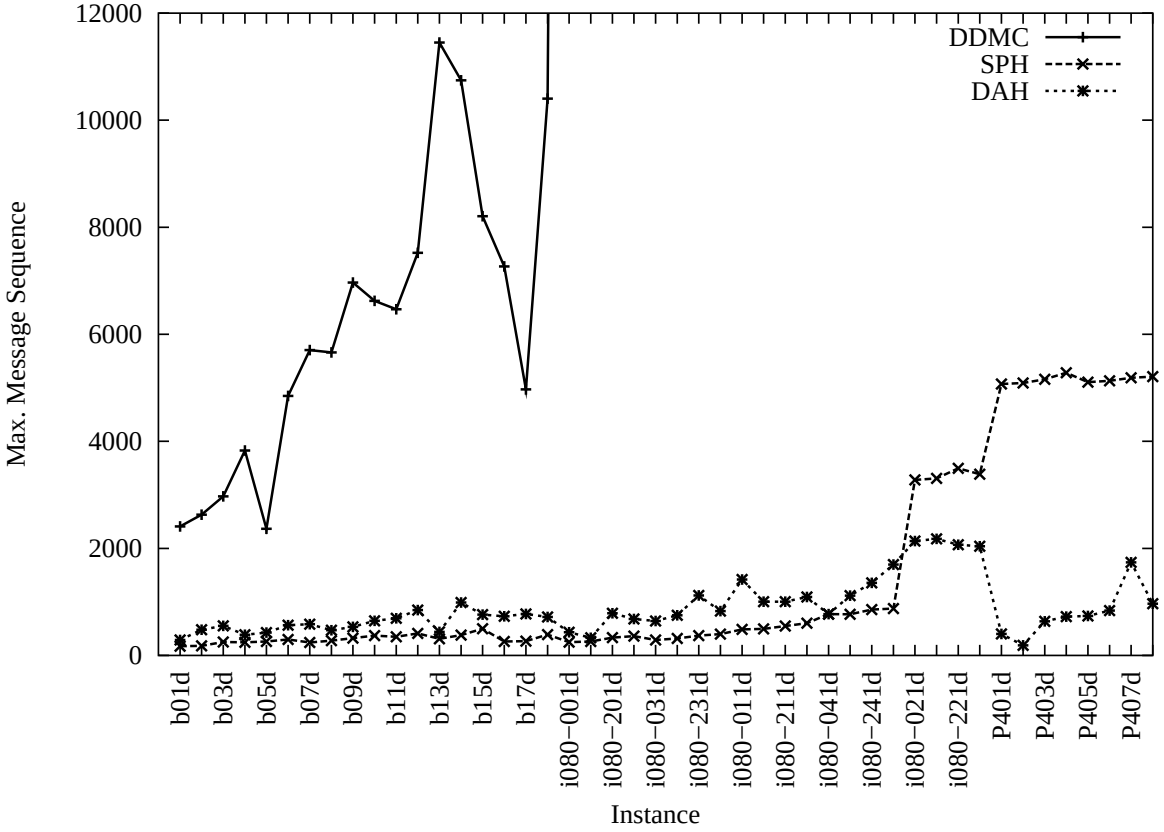

Figure 5: Time – directed instances.

[7] F. Grandoni, J. Knemann, A. Panconesi and M. Sozio, Primal-dual based distributed algorithms for vertex cover with semi-hard capacities, Proc. of the 24th ACM SIGACT-SIGOPS (2005) 839–848.

[8] F. Hwang, D. Richards and P. Winter, The Steiner tree problem, Annals of Discrete Mathematics, 53, North-Holland (1992).

[9] T. Koch, A. Martin and S. Voss, SteinLib: an updated library on Steiner Problems in Graphs, ZIB-Report 00-37 (2000) `http://elib.zib.de/steinlib`.

[10] R. Novak, J. Rugelj and G. Kundus, Steiner tree based distributed multicast routing in networks, In: Steiner Trees in Industries, D.-Z. Du, X. Cheng (Eds.), Kluwer (2001) 327–352.

[11] R. Novak, J. Rugelj and G. Kundus, A note on distributed multicast routing in point-to-point networks, Computers and Operations Research, 26 (2001) 1149–1164.

[12] C. Oliveira and P. Pardalos, A Survey of combinatorial optimization problems in multicast routing, Computers & Operations Research, 32 (2005) 1953–1981.

[13] M. Poggi de Aragão, E. Uchoa and R. Werneck, Dual heuristics on the exact solution of Large Steiner Problems, Electronic Notes in Discrete Mathematics, 7 (2001) 46–51.

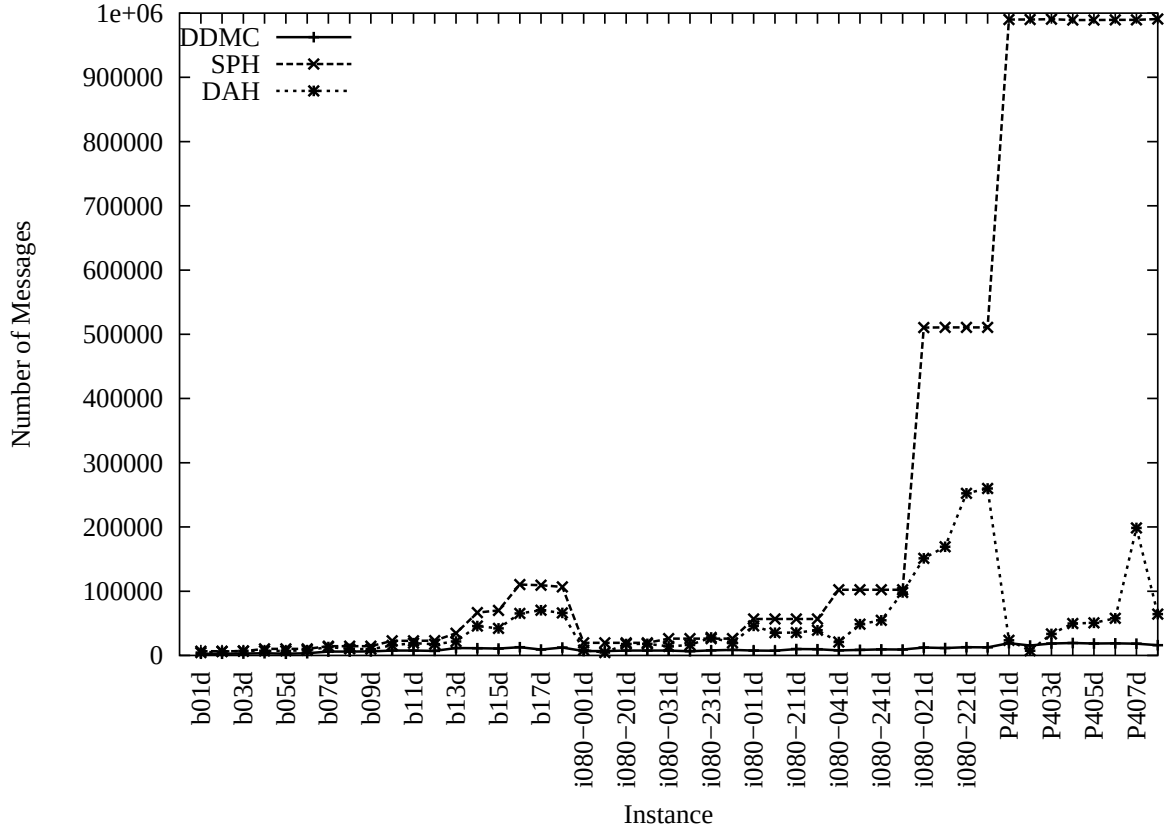

Figure 6: Messages – directed instances.

[14] M. Poggi de Aragão and R. Werneck, On the implementation of MST-based heuristics for the Steiner problem in graphs, Proc. of the 4th ALENEX, Lecture Notes in Computer Science, 2409 (2002) 1–15.

[15] T. Polzin and S. Vahdati, Improved algorithms for the Steiner problem in networks, Discrete Applied Mathematics, 112 (2001) 263–300.

[16] J. Rugelj and S. Klavzar, Distributed multicast routing in point-to-point networks, Computers and Operations Research, 24 (1997) 521–527.

[17] A. Segall, Distributed network protocols, IEEE Transactions on Information Theory, 29 (1983) 23–35.

[18] A. Shaick and K. Shin, Destination-driven routing for low-cost multicast, IEEE Journal on Selected Areas in Communications, 15 (1997) 373–381.

[19] G. Singh and K. Vellanki, A distributed protocol for constructing multicast trees, Proc. of the 2nd International Conference on Principles of Distributed Systems (1998) 61–76.

[20] S. Voss, Steiner's problem in graphs: heuristic methods, Discrete Applied Mathematics, 40 (1992) 45–72.

[21] R. Werneck, Steiner Problem in Graphs: primal, dual, and exact algorithms, Master's Thesis, Catholic University of Rio de Janeiro (2001).

[22] R. Wong, A dual ascent approach for Steiner Tree Problems on a directed graph, Mathematical Programming, 28 (1984) 271–287.

A Algorithm Pseudocode

The following algorithm is executed by each node and consists of a set of responses to each type of message that can be generated. In addition, the response to a spontaneous awakening of the terminal is given. Each node queues incoming messages and response to them in first-come, first-served order. Each fragment has its *leader* node, all other non-root nodes are *non-leader* with respect to that fragment. We assume that for each arc $k = (i, j)$ from node i to j exists another one $k' = (j, i)$ from node j to i. Nodes i and j are said to be neighbors.

Variables maintained at each node for every fragment t to which it belongs:

- **fragment_state(t)**: [active, finished, suspended]. State of fragment **t**.
- **state(t, arc)**: [internal, entering, outgoing, to_leaf, to_leader]. State of the arc **arc** in the fragment **t**. Initialized with entering or outgoing, according to the direction of the arc.
- **subtree_cost(t, i)**: Smallest cost among all entering arcs to fragment **t** in the subtree rooted in the node **i**.
- **conflict(t, i)**: Boolean. If TRUE, indicates that fragment **t** has a conflict in the subtree rooted in the node **i**.
- **end(t, i)**: Boolean. If TRUE, indicates that the root was included by the fragment **t** into the subtree rooted in the node **i**.
- **waited_ack(t)** and **waited_conv(t)**: Number of expected ACK or CONV messages to initiate a convergecast or a broadcast in fragment **t**. Initialized with zero.
- **wait(t)**: Boolean. If TRUE, indicates that fragment t is waiting for the suspension of another fragment with smaller identification number to continue its execution. Initialized with FALSE.
- δ: Smallest cost among all entering arcs in a part of a fragment.

Each fragment leader also maintains the variable **pb** containing a partial bound and Δ containing the smallest cost among all entering arcs in a fragment. The root maintains **terminals** with the number of finalized terminals, **gb** containing the global lower bound and **terminated** that is set to TRUE when all fragments are finished, indicating the end of the algorithm.

Algorithm:

(A.1) Spontaneous awakening, occurs in every terminal t, except the root:

```
t ← terminal id;
Δ ← smallest cost among all entering arcs;
pb ← Δ;
fragment_state(t) ← active;
for each arc k with state(t,k)=entering :
    reduced_cost(k) ← reduced_cost(k) - Δ;
    if reduced_cost(k) = 0
        waited_conv(t) ← waited_conv(t) + 1;
        state(t,k') ← to_leaf;
        send INCLUDE(t) on arc k';
```

(A.2) Response to receipt of INCLUDE(t) on arc k=(i,j):

```
if node is root
    send CONV(t, END) on arc k';
else
    if fragment_state(t) ∈ {active, suspended, finished}
        state(t,k) ← internal;
        send ALREADYINCLUDED(t) on arc k';
    else
        fragment_state(t) ← active;
        state(t,k') ← to_leader;
        for each arc w with state(t,w)=outgoing:
            waited_ack(t) ← waited_ack(t) + 1;
            send CHECK(t) on arc w;
     if waited_ack(t) = 0 execute procedure convergecast(t);
```

(A.3) Response to receipt of ALREADYINCLUDED(t) on arc k=(i,j):

```
state(t,k) ← internal;
waited_conv(t) ← waited_conv(t) - 1;
if waited_conv(t) = 0 execute procedure convergecast(t);
```

(A.4) Response to receipt of CHECK(t) on arc k=(i,j):

```
if fragment_state(t) = active
    state(t,k) ← internal;
    send ACK(t, true) on arc k';
else
    send ACK(t, false) on arc k';
```

(A.5) Response to receipt of ACK(t, answer) on arc k=(i,j):

```
if answer
    state(t,k) ← internal;
else
    if reduced_cost(k)=0
        waited_conv(t) ← waited_conv(t) + 1;
        state(t,k') ← to_leaf;
```

```
7         send INCLUDE(t) on arc k';
8  waited_ack(t) ← waited_ack(t) - 1;
9  if waited_ack(t) = 0 and waited_conv(t) = 0 execute procedure convergecast(t);
```

(A.6) Response to receipt of CONV(t, type, δ) on arc k=(i,j):

```
1  if type = REACTIVATE
2     conflict(t,i) ← FALSE;
3     if ∄t1 > t such that fragment_state(t1)=active
4        if node is the leader of t
5           for each arc w with state(t,w)= to_leaf:
6              waited_conv(t) ← waited_conv(t) + 1;
7              send BROAD(t, REACTIVATE) on arc w;
8        else
9           send CONV(t, REACTIVATE) on arc w such that state(t,w)= to_leader;
10 else if type = SMALLEST
11    subtree_cost(t,i) ← δ;
12 else if type = SUSPEND
13    conflict(t,i) ← TRUE;
14 else if type = END
15    end(t,i) ← TRUE;
16 waited_conv(t) ← waited_conv(t) - 1;
17 if waited_conv(t) = 0 and waited_ack(t) = 0 execute procedure convergecast(t);
```

(A.7) Procedure convergecast(t):

```
1  if ∃t1 < t such that fragment_state(t1)=active
2     wait(t) ← TRUE;
3  else
4     if ∃t1 > t such that fragment_state(t1)=active
5        local_conflict ← TRUE;
6     if node is the leader of t
7        execute procedure leader-broadcast(t);
8     else
9        execute procedure non-leader-convergecast(t);
10    if ∃t1 > t such that wait(t1)=TRUE
11       wait(t1) ← FALSE;
12       execute procedure convergecast(t1);
```

(A.8) Procedure leader-broadcast(t):

```
1  if end(t,v) = TRUE for at least one neighbor v;
2     send BROAD(t, END, pb) on each arc w such that state(t,w)=to_leaf ;
3  else if local_conflict or conflict(t,v) = TRUE for at least one neighbor v;
4     send BROAD(t, SUSPEND) on each arc w such that state(t,w)=to_leaf;
5  else
6     Δ ← smallest cost among
           every arc w such that state(t,w)=entering and
           every subtree_cost(t,v) such that reduced_cost(arc(j,v))=0;
```

```
7     pb ← pb + Δ;
8     for each arc w with state(t,w)=to_leaf:
9        waited_conv(t) ← waited_conv(t) + 1;
10       send BROAD(t, SMALLEST, Δ) on arc w;
11    for each arc w with state(t,w)=entering:
12       reduced_cost(w) ← reduced_cost(w) - Δ;
13       if reduced_cost(w) = 0
14          state(t,w') ← to_leaf;
15          waited_conv(t) ← waited_conv(t) + 1;
16          send INCLUDE(t) on w';
```

(A.9) Procedure non-leader-convergecast(t):

```
1  if end(t,v) = TRUE for at least one neighbor v;
2     send CONV(t, END) on arc w such that state(t,w)=to_leader;
3     fragment_state(t)← finished;
4  else if local_conflict or conflict(t, v) = TRUE for at least one neighbor v;
5     send CONV(t, SUSPEND) on arc w such that state(t,w)=to_leader;
6     fragment_state(t)← suspended;
7  else
8     δ ← smallest cost among
           every arc w such that state(t,w)=entering and
           every subtree_cost(t,v) such that reduced_cost(arc(j,v))=0;
9     send CONV(t, SMALLEST, δ) on arc w such that state(t,w)=to_leader;
```

(A.10) Response to receipt of BROAD(t, SMALLEST, Δ) on arc k=(i,j);

```
1  for each arc w with state(t,w)= to_leaf
2     waited_conv(t) ← waited_conv(t) + 1;
3     send BROAD(t, SMALLEST, Δ) on arc w;
4  for each arc w with state(t,w)=entering:
5     reduced_cost(w) ← reduced_cost(w) - Δ;
6     if reduced_cost(w)=0
7        state(t,w') ← to_leaf;
8        waited_conv(t) ← waited_conv(t) + 1;
9        send INCLUDE(t) on w';
10 if waited_conv(t) = 0 execute procedure convergecast(t);
```

(A.11) Response to receipt of BROAD(t, SUSPEND) on arc k=(i,j);

```
1  fragment_state(t)=suspended;
2  send BROAD(t, SUSPEND) on each arc w such that state(t,w)=to_leaf;
3  if ∃ t_1 > t such that wait(t_1) = TRUE
4     wait(t_1) ← FALSE;
5     execute procedure convergecast(t_1);
```

(A.12) Response to receipt of BROAD(t, REACTIVATE) on arc k=(i,j);

```
1  fragment_state(t)=active;
2  for each arc w with state(t,w)= to_leaf
```

```
3    waited_conv(t) ← waited_conv(t) + 1;
4    send BROAD(t, REACTIVATE) on arc w;
5 for each arc w with state(t,w)=entering and with reduced_cost(w)=0
6    state(t,w') ← to_leaf;
7    waited_conv(t) ← waited_conv(t) + 1;
8    send INCLUDE(t) on arc w';
9 if waited_conv(t) = 0 execute procedure convergecast(t);
```

(A.13) Response to receipt of BROAD(t, END, lower_bound) on arc k=(i,j);

```
1 fragment_state(t)=finished;
2 if node is not root
3    send BROAD(t, END, lower_bound) on each arc w such that state(t,w)= to_leaf;
4    if ∃ t1 > t such that wait(t1) = TRUE
5       wait(t1) ← FALSE;
6       execute procedure convergecast(t1);
7    else if ∃ t1 such that t1 is the largest fragment identification with fragment_state(t1)=suspended
8       send CONV(t1, REACTIVATE) on arc w such that state(t1,w)= to_leader;
9 else
10    gb ← gb + lower_bound;
11    terminals ← terminals + 1;
12    if terminals = total number of terminals
13       terminated ← TRUE
```

B Example of Execution

We illustrate a possible algorithm execution over a small instance with only four nodes, three terminals and a non-terminal, shown in Figure 7. Arcs are represented by dashed arrows, arc reduced costs by the numbers beside the arrows, terminals by squares and the non-terminal by a circle. We also show the partial lower bounds, **pb**, of the non-root terminals and the root global lower bound, **gb**.

Figure 8 presents the graph after the execution of actions (A.1), when each non-root terminal awakes spontaneously, subtracts the smallest cost from all entering arcs and sends *Include* messages on saturated arcs that are represented by solid arrows.

Figure 9 shows the execution of (A.2) by node 3, when it sends *Check* messages to its neighbors. Figure 10 presents the *Ack* messages generated when t_1, t_2 and R execute (A.4).

Upon receiving an *Ack* message, node 3 executes (A.5). When it receives the last *Ack* message, it executes the procedure convergecast (A.7). As node 3 is a non-leader, the procedure non-leader-convergecast, (A.9), is executed. It detects a conflict and sends a *Conv* message to each fragment leader. The fragment leader t_1 receives a *Conv(Suspend)* message while t_2 receives a *Conv* message containing the smallest cost, as shown in Figure 11.

In Figure 12, t_1 informs the suspension of the fragment through a *Broad(Suspend)* message ((A.8), lines 3 and 4), and t_2 initiates a new growing round sending a *Broad* message ((A.8), lines 6 to 10) and an *Include* message ((A.8), lines 11 to 16).

In Figure 13, node 3 sends an *Include* message to the root. Note that in this example this node receives two *Include* messages in the same growing round. A non-root node would answer

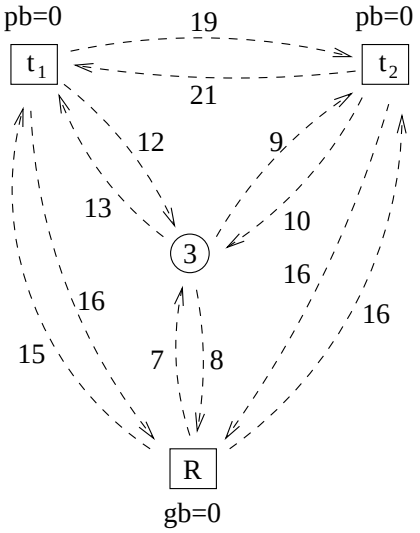

Figure 7: Original Graph

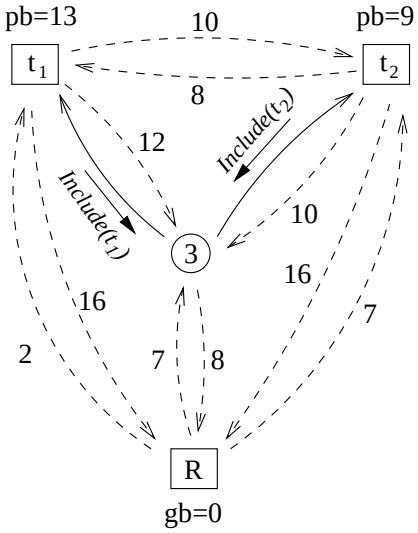

Figure 8: Including node 3

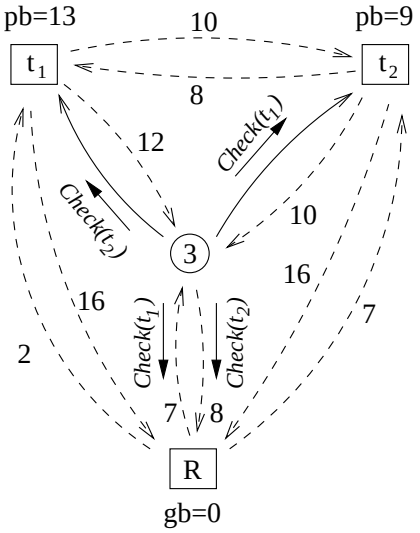

Figure 9: Checking neighbors

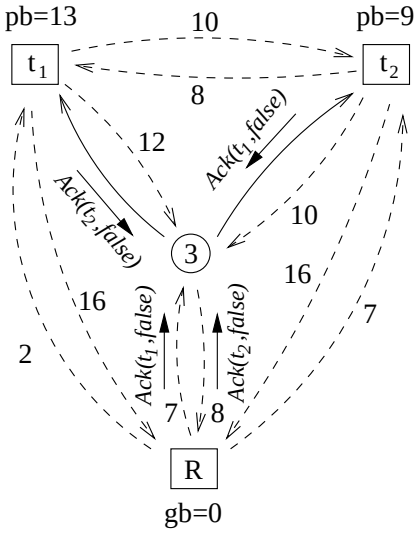

Figure 10: Sending *Ack* messages

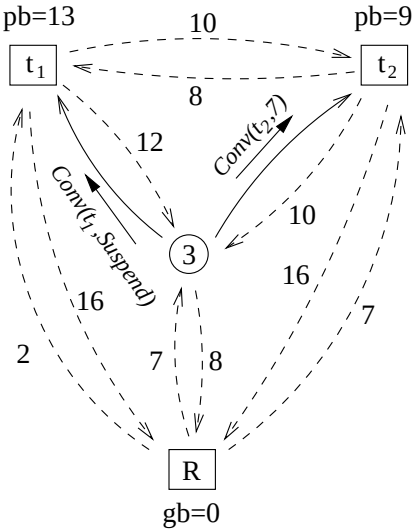

Figure 11: Convergecasts to suspend fragment t_1 and with the smallest cost to fragment t_2

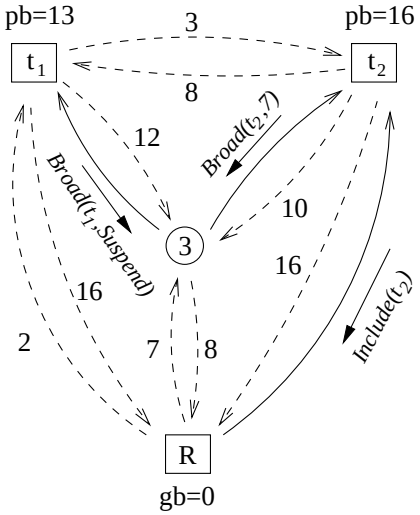

Figure 12: Suspending fragment t_1 and Growing fragment t_2

with an *AlreadyIncluded* message to the node from which it received the last *Include* message ((A.2), lines 4 to 6). A root node answers with a *Conv(End)* message ((A.2), line 2), as shown in figures 13 and 14.

In Figure 15, t_2 initiates the broadcast of *Broad(End)* ((A.8) lines 1 and 2). Upon receiving this message, the root updates the global partial bound ((A.13) line 10) and node 3 tries to reactivate fragment t_1 ((A.13), lines 7 and 8), as can be seen in Figure 16.

In Figure 17, when node 3 is reactivated by a *Broad(Reactivate)* message, it sends an *Include* message on the arc previously saturated by fragment t_2.

In Figure 18, we can observe the messages sent to finish the fragment t_1 and in Figure 19, the broadcast of t_1's partial lower bound. At this point, the root detects that all fragments are finished.

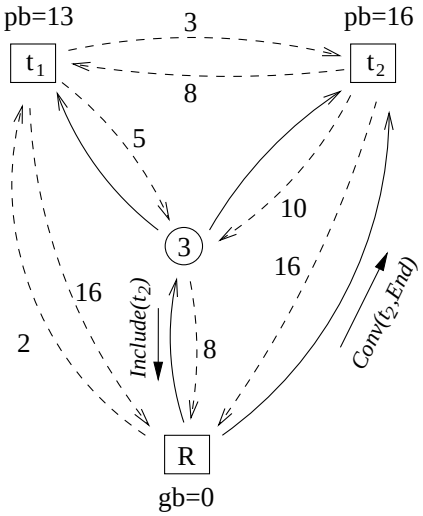

Figure 13: Root is reached by fragment t_2

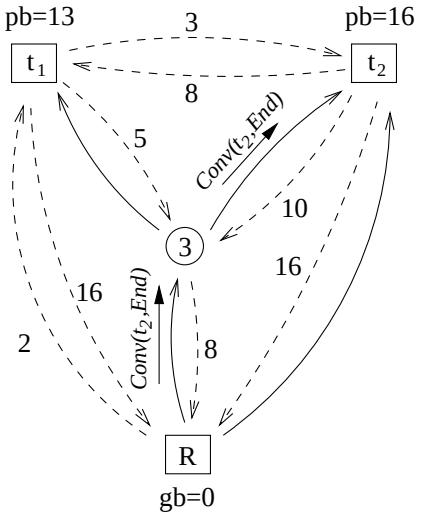

Figure 14: Convergecast to stop fragment t_2

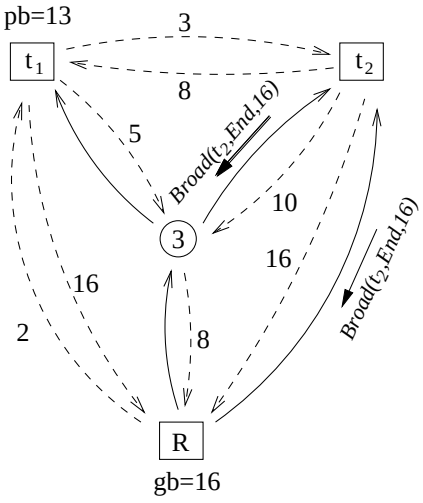

Figure 15: Finishing fragment t_2

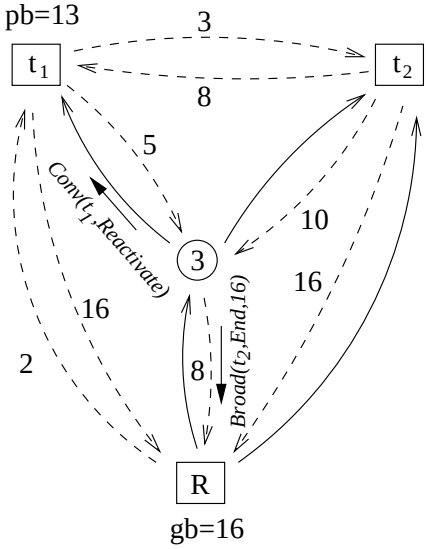

Figure 16: Reactivating fragment t_1

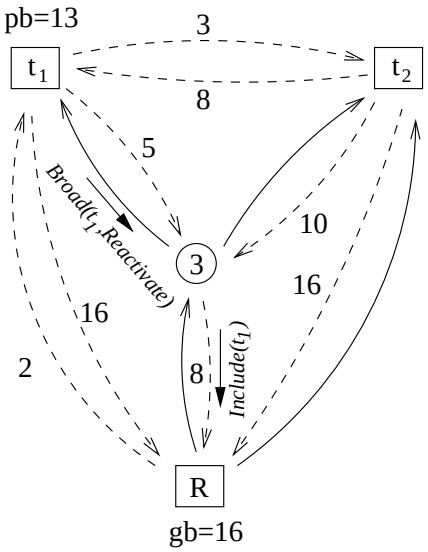

Figure 17: Root is reached by fragment t_1

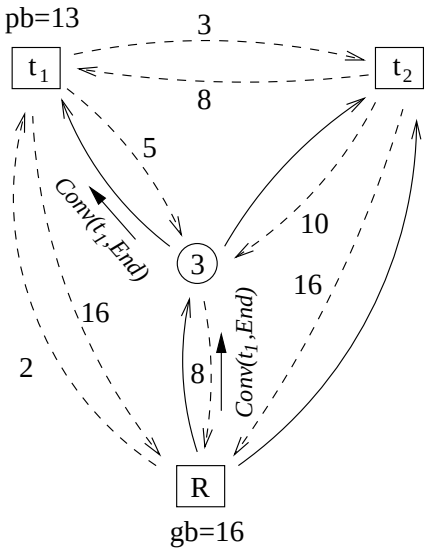

Figure 18: Convergecast to stop fragment t_1

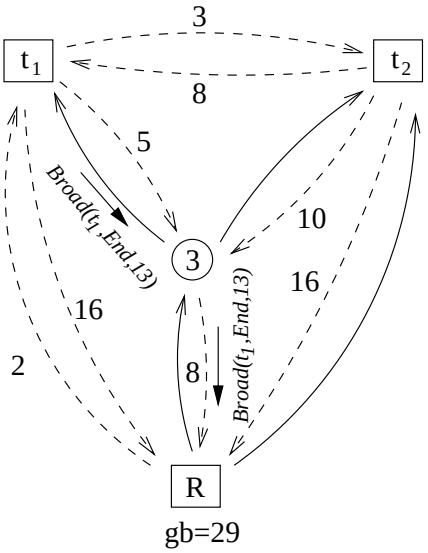

Figure 19: Fragment t_1 finishes and root detects termination