Distributed Dual Ascent Algorithm for Steiner Problems in Networks

Marcelo Santos[1], Lúcia M. A. Drummond[1], Eduardo Uchoa[2]

[1]Department of Computer Science

[2]Department of Production Engineering
Fluminense Federal University – Niterói, Brazil

{mpinto,lucia}@ic.uff.br, uchoa@producao.uff.br

Abstract. *Steiner Problems in undirected or directed graphs are often used to model multicast routing problems. The directed case being particularly suitable to situations where most of the trafic has a single source. Sequential Steiner heuristics are not convenient in that context, since one can not assume that a central node has complete information about the topology and the state of a large wide area network. This work presents a distributed version of the Dual Ascent Heuristic proposed by Wong, known for its remarkable good practical results, lower and upper bounds, in both undirected and directed Steiner problems. The distributed Dual Ascent has worst case complexities of $O(|V|^2)$ time and $O(|T|.|V|^2)$ messages. Experimental results are also presented, showing the eficiency of the proposed algorithm.*

1. Introduction

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 problem is often modelled as a Steiner Problem in Graphs (SPG), as surveyed by [Novak et al. 2001b] and also by [Oliveira and Pardalos 2005]. 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 modelled as a Steiner Problem in Directed Graphs (SPDG).

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) is the case where $G_D = (V, A)$ is a directed graph and there is a special root terminal $r \in T$. 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 if solutions must be obtained in short time.

Sequential Steiner heuristics are not much suitable for multicast routing, since one can not 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 each node initially only knows about its immediate neighborhood.

Some distributed SPG heuristic algorithms [Chen et al. 1993] utilize previous distributed algorithms for the Minimum Spanning Tree (MST) problem [Gallager et al. 1983]. They first build a MST and then execute a prune phase, removing subtrees that do not contain terminals. This simple algorithm may lead to poor solutions. There is another drawback: all network nodes are involved in the computation, even when only a few of them are to be connected. It is desirable for a distributed algorithm over networks to be locality-sensitive, the computation effort should decrease when the nodes relevant to the solution are clustered. More sophisticated algorithms in the literature are distributed versions of the Shortest Path Heuristics (SPH). The Prim-SPH (a.k.a. Cheapest Insertion Heuristic) 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. Its distributed versions [Bauer and Varma 1996, Rugelj and Klavzar 1997] construct, in parallel, shortest paths from each node to each non-root terminal. Those shortest paths are used by another parallel thread, that starts from the root to build the Steiner Tree. The time complexity of those algorithms, measured by the maximum sequence of messages, is $O(|T|.|V|)$. The overall number of exchanged messages is $O(|V|^2)$. [Novak et al. 2001a] proposed improvements on those algorithms leading to a better practical performance, but could not change the worst case complexities. The above mentioned distributed Prim-SPH algorithms are locality-sensitive and can be adapted to the SPDG.

The so-called Kruskal-SPH (although it actually resembles Borüvka's MST algorithm) grows several subtrees at once, starting at each terminal. At each step some pairs of subtrees are joined by shortest paths. Its distributed version was proposed by [Bauer and Varma 1996], 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 1998]; this also improves the time complexity when $|T|$ is not $O(\log |V|)$. Those algorithms (or even the sequential Kruskal-SPH) can not 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 non-terminal node with minimum average distance to each subtree. The distributed version by [Gatani et al. 2005] takes $O(|T|.|V|)$ time and $O(|E| + |T|.|V|)$ messages. The ADH can not be adapted to the SPDG.

Those distributed algorithms (Prim-SPH, Kruskal-SPH and ADH) assume that each node already knows its shortest distance to all other nodes. If this is not the case, the distributed computation of such distances would add a message complexity of $O(|E|.|V|)$, a bit complexity of $O(|E|.|V|. \log |V|)$ and a time complexity of $O(|V|)$ [Segall 1983].

The SPG and SPDG algorithm proposed in this article is a distributed version of the sequential dual ascent algorithm proposed by [Wong 1984]. This algorithm has the following advantages over other heuristics.

- Extensive computational experiments over the main classes of SPG benchmark instances from the literature have shown that Dual Ascent usually

yields better solutions [Voss 1992, de ARAGÃO et al. 2001, Werneck 2001, Polzin and Vahdati 2001, de ARAGÃO and Werneck 2002] than Prim-SPH. Kruskal-SPH and ADH are a little worse than Prim-SPH.

- The Dual Ascent is an example of what was latter called a *primal-dual algorithm* [Goemans and Williamson 1996], it can be interpreted as working in the dual of a linear program formulation. In practice, this means that Dual Ascent not only returns a solution, it also returns a guarantee of the quality of this solution. For example, it may yield a solution of value 1000 together with a lower bound of 980 on the cost of any other solution. This means that this particular solution is guaranteed to be within 2% away from the optimal. The quality of Dual Ascent lower bounds is remarkable, they are usually less than 3% bellow the optimal. For this reason, Dual Ascent is a key part of the best exact algorithms for the SPG [Polzin and Vahdati 2001, de ARAGÃO et al. 2001].
  Tight lower bounds can be very useful. If the user is not satisfied with the guarantee obtained, he may want to run the Dual Ascent again (this distributed algorithm is not deterministic), or any other heuristic, trying to get better solutions. Moreover, the lower bounds can be used to remove arcs from the instance, by proving that they do not belong to an optimal solution. It is typical to remove more than half of the arcs. A second run of Dual Ascent (or of any good heuristic) on that reduced instance is quite likely to improve the solution.
- It does not require that nodes know the value of least cost paths to every other node. Some authors [Bauer and Varma 1996, Novak et al. 2001a, Gatani et al. 2005] argue that network layer protocols like RIP and OSPF already keep routing tables with that information, it could 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 appropriated 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. Of course, one can always run a shortest distance algorithm with respect to the desired costs before computing the tree. However, this would add the above mentioned complexities [Segall 1983] to the overall method and it is not locality-sensitive.

The proposed distributed Dual Ascent has worst case complexities of $O(|V|^2)$ global time, $O(|T|.|V|^2)$ messages and $O(|V|)$ local time complexity. One alternative to perform the Dual Ascent 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 Dual Ascent algorithm is $O(|E|^2)$. Considering such complexities and the memory demand at the leader node, the fully distributed approach is an appealing alternative for multicast routing.

The remainder of this paper is organized as follows. Section 2 introduces the

sequential Wong's Dual Ascent Algorithm. The distributed algorithm is presented and analyzed in Section 3. Section 4 presents experimental results, comparing the practical performance of our algorithm with Prim-SPH in terms of solution quality, time and exchanged messages.

2. Sequential Wong's Dual Ascent Algorithm

Wong's Dual Ascent is an algorithm for the SPDG. However, a simple transformation allows its effective use on the SPG too. In this case, define $G_D = (V, A)$ to be the directed graph obtained by replacing each edge in E by two opposite arcs and choosing any terminal r to be the root. 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 Dual Ascent, 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 cost are positive, this graph starts with no arcs. All operations in the algorithm 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 set containing a directed cycle passing by any pair of nodes in it. This set is a *root component* if (i) R contains a terminal, (ii) R does not contains 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 algorithm follows:

Wong's Dual Ascent
$\bar{c}_a \leftarrow c_a$, for all $a \in A$; $LB \leftarrow 0$;
While (exists root components 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: $\bar{c}$ and LB

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 incident to $W(R)$ 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. Therefore it contains at least one SPDG solution. But G_S also contains several redundant arcs. Wong proposed the following additional cleaning steps to obtain good solutions:

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 [Polzin and Vahdati 2001, de ARAGÃO et al. 2001] found that running Prim-SPH over the sparse graph G_S is very fast and yields solutions that are even better than those provided by the MST-and-prune cleaning method proposed by Wong. Anyway, the final output obtained after the cleaning is a solution, together with the lower bound LB, which gives a guarantee of the quality of this solution. The overall complexity is dominated by the Dual Ascent step, $O(|E|^2)$. The reduced costs provided by the Dual Ascent step may still be very useful. Let UB be the cost of a solution. It can be shown that all arcs a such that $LB + \bar{c}_a \geq UB$ can not belong to any solution with cost smaller than UB. This means that all such arcs should be eliminated from any attempt of searching for improving solutions.

Extensive experiments have shown that the guarantees provided by Dual Ascent are very good in practice [Voss 1992, de ARAGÃO et al. 2001, Polzin and Vahdati 2001, Werneck 2001, de ARAGÃO and Werneck 2002]. However, this is not an approximative algorithm, since this guarantee is not limited by any constant factor. In fact, no algorithm approximating the SPDG by a constant factor is possible unless $P = NP$.

3. The Distributed algorithm

We propose an asynchronous distributed version of the previously described algorithm. Its output is the lower bound and the arc reduced costs, which also gives the saturation graph. The cleaning steps to actually compute a solution from this graph can be performed by already known distributed algorithms. We assume that each node knows the cost of each arc incident to 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 nodes of the graph have distinct identities that can be totally ordered.

The agents in this distributed algorithm are associated to the *fragments*, defined as a set $W(t)$ formed by all nodes reaching a non-root terminal t. The terminal t identifying a fragment $W(t)$ is also known as its *leader*. Note that if t belongs to a strongly connected component R, $W(t) = W(R)$. Strongly connected components are disjoint by definition, but sets $W(R)$ are not. Therefore, a node can work to several fragments, perhaps belonging to different connected components.

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 leader to nodes. A fragment *growing round* consists of finding the minimum reduced cost of an incident arc and subtracting this value from the reduced cost of all incident arcs. The first operation is performed using a convergecast, the result is then broadcasted. The decreasing of reduced costs causes some new arcs to be saturated, the corresponding nodes must be included in the fragment and the fragment tree 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 incident arcs cre-

ates a mutual exclusion problem, that causes some fragments to suspend their operations temporarily. A fragment should stop its execution when it is reached from the root. When the root reaches all terminals, it initiates a broadcast to terminate the algorithm.

3.1. Growing Fragments

At first, a fragment composed by just a terminal t selects the incident 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 incident 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 in 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, *Ack(t,y)* if the neighbor also belongs to t and *Ack(t,n)* if it does not.

The next step is calculating the minimum reduced cost of all arcs incident to the fragment. A leaf node in the fragment sends by its *To_leader(t)* arc a message *Conv(t,Smallest,Δ)* with the minimum reduced cost of an arc incident to it, Δ, and not belonging to the fragment. The internal nodes in the fragment, upon receiving its *Conv(t,Smallest,Δ)* messages, compare those costs with the minimum reduced cost of an arc incident to it and not belonging to the fragment. The smallest such value is sent by its *To_leader(t)* arc using another *Conv(t,Smallest,Δ)* message. When the fragment leader finally knows the fragment minimum reduced cost, it updates the fragment lower bound and starts broadcasting this value back to all fragment nodes, using *Broad(t,Smallest,Δ)* messages. Upon receiving such messages, a node decreases the cost of its incident arcs, which may trigger *Include(t)* messages on the newly saturated arcs, therefore growing the fragment. The newly added nodes 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.

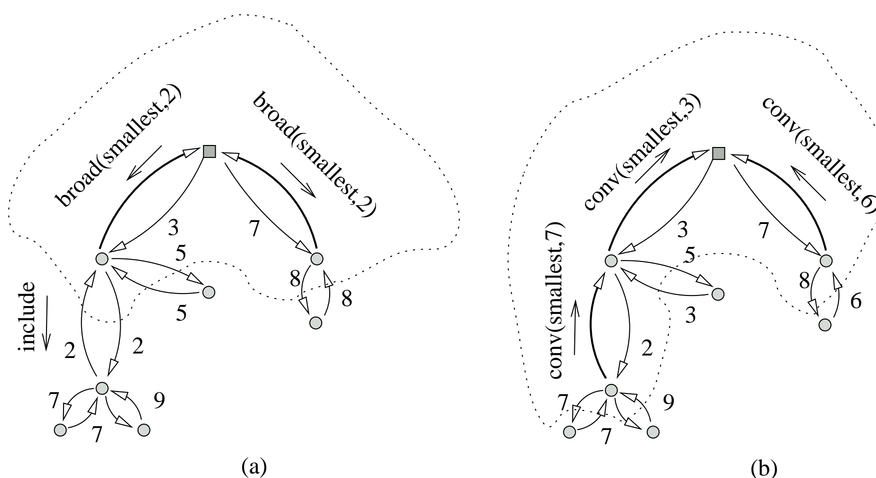

Figure 1. Growing Fragments

When several nodes are included in a fragment in the same growing round, it is possible that a newly added node i receives a *Ack(t,n)* from a node j that will still receive a *Include(t)* in that round. This means that i will consider arc (j, i) as incident to the fragment, it is possible that its reduced cost reaches the leader as the smallest. When a *Broad(t,Smallest,*Δ*)* with this value reaches i, this node will be already informed (by a *Check(t)* message) that j is actually part of the fragment. Arc (j, i) will be saturated but no *Include(t)* message will be sent by it. Therefore, it is possible to have *degenerated growing rounds*, i.e. rounds where only arcs internal to the fragment are saturated and no new node is included. However, it is not possible to have two degenerated growing rounds in sequence.

Another situation happens when nodes i and j send *Include(t)* messages to the same node k. Suppose that the message from i arrives first. The second *Include(t)* message should be answered by a *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 incident arc not belonging 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 (i, j) 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.

3.2. Suspending and Stoping Fragments

Fragments with common incidents arc can not grow at the same time, since the reduced cost of those arcs would be changed concurrently. Here we have a classical mutual exclusion problem, where the shared resources are the common incident arcs. In order to solve this conflict, only the fragment with the largest identification will go on growing, while the other is suspended. It remains suspended until the fragment in conflict finishes its execution or is suspended by another fragment. When a node that already belongs to some fragments receives an include message from another fragment, there is a potential conflict. In order to simplify the algorithm, without sacrificing its correctness and complexities, this condition is enough to suspend all but one of the fragments. At a given moment, each node is associated to at most one active fragment.

Two cases should be considered. When a node receives *Include(*t_1*)* from a fragment with identification smaller than its current active fragment t_2, it answers *Conv(*t_1*,Suspend)* immediately. On the other case, when the identification of t_1 is greater than t_2, the node must wait t_2 finish its current growing round, before suspending t_2 and continue the growing of t_1.

The leader of a suspended fragment t_2 sends *Broad(*t_2*,Suspend)* messages, indicating to all its nodes that it was suspended. When a conflict node receives such a message indicating that its current active fragment t_2 was suspended, it tries to reactivate other fragments, like t_1, that were suspended by it. This is done by sending to their leaders a *Conv(*t_1*,Reactivate)* message. In case other conflicts do not exist, the leader of t_1 will send *Broad(t,Reactivate)* messages to re-initiate its growing. The overall situation is illustrated

in Figure 2. Fragments t_1 and t_2 have a common node j, t_1 is suspended. Then t_2 grows and conflicts with t_3 at node k, t_2 is suspended and re-activates t_1. When a fragment finishes, as will be described, similar reactivation messages may be also sent.

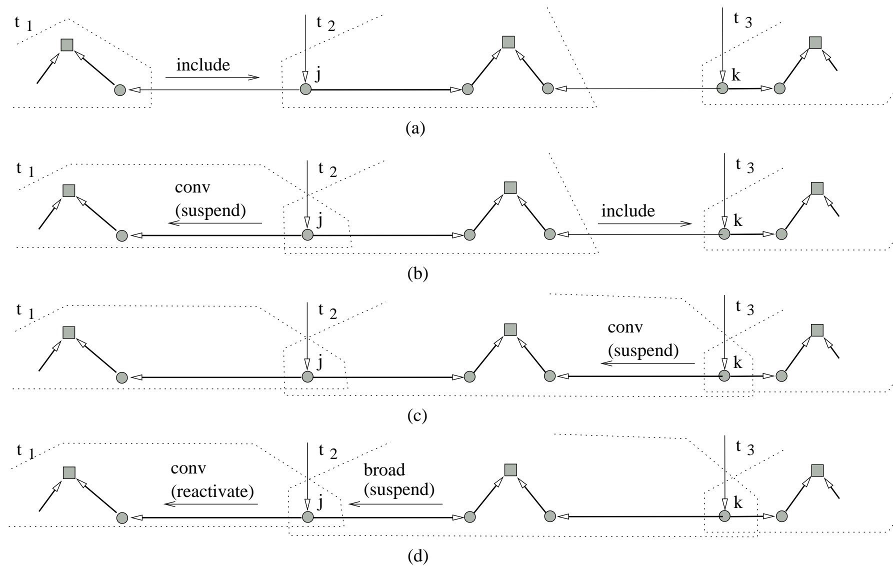

Figure 2. Suspending and Reactivating Fragments

A fragment may stop growing definitively because it can be reached from the root node by saturated arcs. When the root node receives a *Include(t)*, it sends back a *Conv(t,End,root)*. The fragment leader then sends a *Broad(t,End,root,lb)* with its partial lower bound and stops. The root node accumulates those values to obtain the global lower bound. When a fragment is stopped it also tries to reactivate other suspended fragments in conflict with it, as occurs when a fragment is suspended.

When the root reaches all terminals, it initiates a broadcast to terminate the algorithm.

3.3. Algorithm Analysis

3.3.1. Correctness

Lemma 1 *The algorithm is deadlock-free.*

Proof: A fragment executes until it is suspended or stopped. Considering suspended fragments, by the description in Subsection 3.2, it follows immediately that mutual exclusion is guaranteed in the access to incident arcs of a shared node and that in such conflict the one with the largest identification goes on executing. For this reason at least one fragment will succeed in executing avoiding a wait cycle of suspended fragments and consequently the deadlock. ∎

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

Proof: By the description of Subsection 3.1, for each two waves of broadcast, at least one node is included in a fragment. The fragment executes, until it is reached by the root or until it is suspended. In the second case, let us suppose that a fragment, t_i, suspends

its processing because it shares a node with another active fragment t_j such that $t_j > t_i$. Let us still suppose that t_i remains suspended permanently. By Lemma 1, t_j will succeed in executing. By description of Subsection 3.2, it will execute until being suspended or stopped, and in both cases, it will try to reactivate other possibly remaining suspended components, including t_i. At least one of them, the one with the largest identification, will succeed. This procedure will be repeated successively, allowing for all suspended components to continue its execution. Then, t_i will be reactivated in a finite time and will continue the execution until being reached by the root. So, a contradiction exists with the possibility of a fragment is never reached by the root. ∎

3.3.2. Communication and Time Costs

We want an upper bound on the number of messages exchanged during the execution of the algorithm. The worst case would occur in a complete graph where the terminals (including the root) are connected by high cost arcs, in such a way they will only be saturated when all non-terminal nodes are already included in all fragments. When a node is included in a fragment, it checks all its $|V|$ neighbors, this will be done $|V| - |T|$ 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 demands $O(|V|)$ *Conv*, *Broad* and *Include* messages. As there can be up to $O(|V|)$ growing rounds by fragment, $O(|T|.|V|^2)$ is a valid upper bound on the number of such messages.

Considering the suspending and reactivation procedures, the worst situation occurs when we have $|T|$ fragments whose leaders, are $t_1, t_2, \ldots, t_{|T|}$, such that t_1 conflicts with t_2 at a node, t_2 conflicts with t_3 at another node and so on. Assume those leaders are ordered in increasingly identification order. Then, t_1 is suspended while t_2 goes on executing. When t_3 suspends t_2, t_1 is reactivated. Then t_4 suspends t_3, t_2 is reactivated and t_1 is suspended again. The total of re-activations is $O(|T|^2)$. For each reactivation, a broadcast on the tree is executed, that will never require more than $O(|V|)$ messages.

The time complexity considers messages that happen sequentially, occurring in all executions of the algorithm. Intuitively, as more active fragments execute, more arcs are saturated in parallel. Although it appears that the algorithm allows an amount of parallelism limited to the number of terminals, there are particular situations where there is almost no parallelism. As in the previous section, the worst case would occur in a complete graph where the terminals (including the root) are connected by high cost arcs, in such a way they will only be saturated when all non-terminal nodes are already included in all fragments.

Let us suppose that we have a fragment where one arc is saturated at each broadcast, and that when it stops, the resulting tree has two subtrees, one that is a unique long chain of saturated arcs and the other containing the root. For each new node included in this fragment, the causal chain of message receiving and sending is increased permanently by two, corresponding to the receiving and the sending of *Broad* and *Conv* messages in the next growing rounds. At the moment a new node is included a causal chain two long, corresponding to the sending of *Check* and the receiving of *Ack* messages, is also formed, but

this chain occurs only in the round the node is included. So the resulting causal chain considering broadcasts, convergecasts and the checks is $O(V^2)$. Remark that the same may occur for only one more fragment. After this, a connected component is formed allowing for all other fragments, in a broadcast wave that includes any node of such component, the incorporation of all other nodes of the component in a linear time. Summarizing, for only two fragments the causal chain is $O(V^2)$, the others will be reached by the root in $O(V)$ time. So, the complexity is $O(V^2)$.

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 256 processors. Half of the tests were performed over the SPG benchmark instances from the SteinLib [Koch et al. 2000]. 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 harder to solve. Instances from P4Z series have complete graphs. All instances available at 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 to perform the remaining tests. 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. Therefore, the cost of arc (i, j) is likely to be different from the cost of (j, i). Results in Table 1 refer to those directed instances. We also implemented the distributed version of the Prim-SPH found in [Novak et al. 2001a] (adapted to digraphs), including a shortest distance algorithm. This Prim-SPH was applied as a stand-alone algorithm and also, as the cleaning step, to the reduced graphs produced by our distributed dual-ascent algorithm.

Columns in Table 1 have the following meaning: $|V|$, $|A|$, and $|T|$ give the instance size; **Opt** is the value of the optimal solution (calculated with the branch-and-bound code from [de ARAGÃO et al. 2001]); **SPHc** is the value of the solution obtained by running Prim-SPH as a stand-alone algorithm; **LB** is the lower bound given by Dual Ascent; next column gives the proportion of the arcs that are saturated; **DA+SPHr** is the value of the solution obtained after the cleaning step, by running the Prim-SPH over the graph containing only the saturated arcs. Remark that the results given for the Dual Ascent are averages of 5 runs. The sequential Dual Ascent may yield different results depending on the order in which the root components are chosen. In its distributed version those choices depend on non-deterministic factors, therefore each run actually produces slightly different results. The mean standard deviation of the 5 solution values obtained for each instance was 0.6%.

We use *competitiveness*, the ratio of the heuristic cost and the optimal cost, to compare the quality of the solutions provided by Dual Ascent and Prim-SPH. Charts in figures 3 and 4 show the cumulative percentage of cases whose competitiveness is less than or equal a given value, for undirected and directed instances. It is clear that DA + SPHr gives better solutions than SPHc. We also charted the improved results of executing both DA + SPHr and SPHc 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 DA + SPHr and execute SPHc only if it exceeds a given limit. We applied this idea, executing SPHc only if (DA + SPHr) / LB

Instance					SPHc	Dual Ascent		DA+SPHr
Name	$\|V\|$	$\|A\|$	$\|T\|$	Opt	Solution	LB	$(\|Ar\|/\|A\|)\%$	Solution
b01d	50	126	9	77	82	77.0	13.9	**77.0**
b02d	50	126	13	101	107	100.4	13.4	**104.0**
b03d	50	126	25	170	176	165.4	10.3	**173.4**
b04d	50	200	9	58	61	57.4	13.5	**59.6**
b05d	50	200	13	61	**64**	60.6	13.6	65.4
b06d	50	200	25	128	134	126.8	13.2	**129.6**
b07d	75	188	13	122	**122**	121.6	13.6	125.0
b08d	75	188	19	115	126	114.8	13.7	**116.0**
b09d	75	188	38	240	244	239.2	13.4	**242.4**
b10d	75	300	13	90	**91**	89.0	13.5	92.6
b11d	75	300	19	103	**104**	100.8	13.1	105.2
b12d	75	300	38	168	**171**	161.0	11.5	174.0
b13d	100	250	17	176	202	163.8	13.3	**179.4**
b14d	100	250	25	250	261	234.0	13.4	**250.4**
b15d	100	250	50	342	356	341.0	13.4	**345.6**
b16d	100	400	17	133	**133**	131.6	13.4	141.6
b17d	100	400	25	139	**142**	137.0	13.3	145.2
b18d	100	400	50	258	271	258.0	13.8	**259.8**
i080-001d	80	240	6	1751	1815	1729.4	16.3	**1780.8**
i080-011d	80	700	6	1220	1296	1220.0	17.4	**1227.6**
i080-021d	80	6320	6	741	847	673.0	5.8	**768.4**
i080-031d	80	320	6	1514	1706	1455.0	16.5	**1552.8**
i080-041d	80	1264	6	946	1026	900.0	5.9	**983.0**
i080-101d	80	240	8	2322	2706	2322.0	10.5	**2322.0**
i080-111d	80	700	8	1580	1600	1499.2	13.2	**1580.2**
i080-121d	80	6320	8	977	1097	910.0	5.3	**1050.0**
i080-131d	80	320	8	1875	2061	1844.0	17.9	**2016.0**
i080-141d	80	1264	8	1404	1596	1265.6	10.1	**1404.0**
i080-201d	80	240	16	4321	4502	4228.4	22.7	**4322.8**
i080-211d	80	700	16	2951	3174	2808.6	12.8	**3141.0**
i080-221d	80	6320	16	1985	2293	1799.0	5.1	**2082.0**
i080-231d	80	320	16	4156	4623	4036.0	14.0	**4415.2**
i080-241d	80	1264	16	2492	**2617**	2351.2	8.7	2746.8
i080-301d	80	240	20	5714	6365	5390.8	24.8	**5916.0**
i080-311d	80	700	20	3471	**3813**	3266.6	12.9	4065.0
i080-321d	80	6320	20	2453	3137	2374.0	4.1	**2755.4**
i080-331d	80	320	20	4506	4911	4258.6	19.3	**4877.0**
i080-341d	80	1264	20	3132	**3361**	3025.0	15.4	3412.0
P401d	100	9900	5	145	165	145.0	0.4	**145.0**
P402d	100	9900	5	102	**102**	102.0	0.2	**102.0**
P403d	100	9900	5	169	186	166.0	0.5	**180.0**
P404d	100	9900	10	270	**279**	214.8	0.3	285.0
P405d	100	9900	10	248	250	243.2	0.5	**248.0**
P406d	100	9900	10	281	303	226.8	0.3	**296.0**
P407d	100	9900	20	546	590	523.0	0.4	**575.0**
P408d	100	9900	20	502	520	480.2	0.3	**502.0**

Table 1. Results for directed instances.

exceeded 1.05, this happened in 49% of the cases. Those results are also charted, in fact, the last two curves are undistinguishable.

We also compare the practical performance of Dual Ascent and Prim-SPH 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 to 10. For the sake of space, we omit similar measurements on undirected instances. It can be seen that Dual Ascent 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, Dual Ascent performed better on larger I080 instances and much better on P4Z instances. This happens because the cost of running Prim-SPH in those instances is dominated by the calculation of shortest distances among all nodes of large graphs. Dual Ascent, being locality-sensitive, only explores part of those graphs. Remark that Dual Ascent also runs Prim-SPH in its cleaning step, but this run is performed over a smaller graph.

The practical performance of Dual Ascent is quite better than its worst-case complexities. Anyway, the experiments confirmed the analytical prediction made in Section 3 that *Ack* and *Check* messages would dominate the Dual Ascent message complexity in dense graphs. Those messages represented 94.7% of the traffic on P4Z instances.

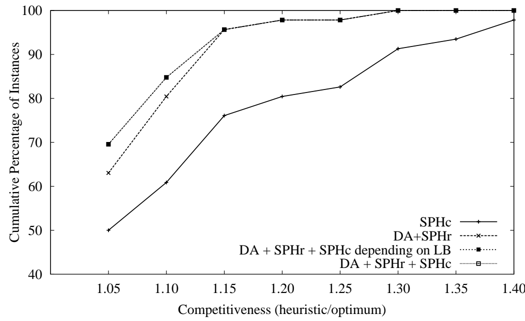

Figure 3. Solution quality – undirected instances.

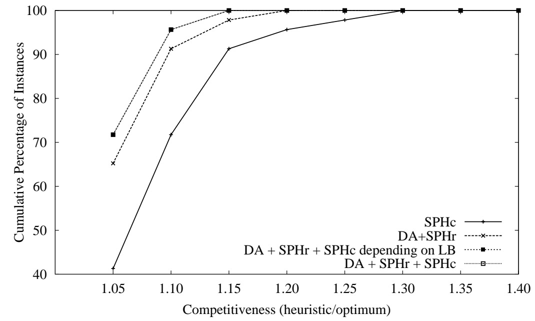

Figure 4. Solution quality – directed instances.

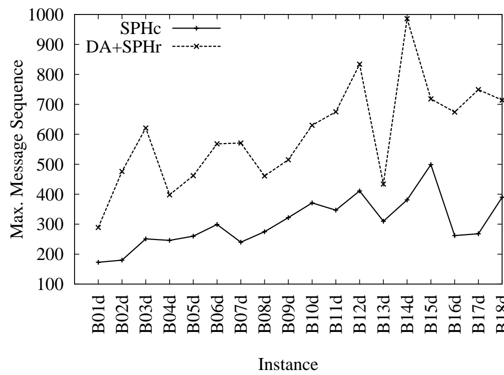

Figure 5. Times – directed B instances.

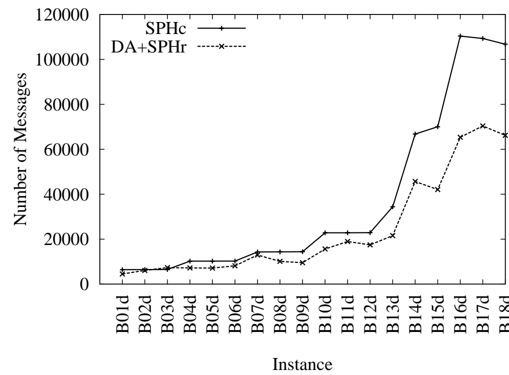

Figure 6. Messages – directed B instances.

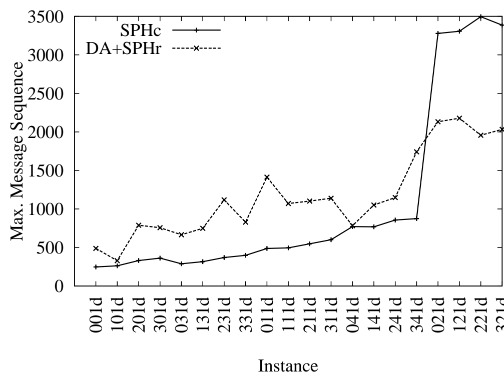

Figure 7. Times – directed I080 instances.

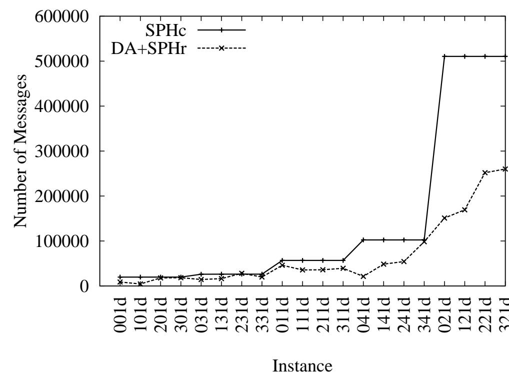

Figure 8. Messages – directed I080 instances.

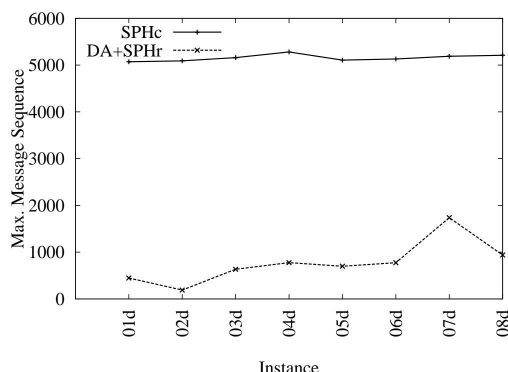

Figure 9. Times – directed P4Z instances.

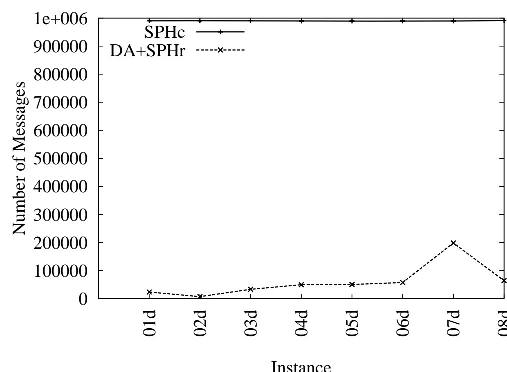

Figure 10. Messages – directed P4Z instances.

5. Conclusion

We presented a distributed version of Wong's DA algorithm for the Steiner problem in graphs, which can be directly applied in multicast routing. The experimental evaluation of this algorithm, comparing its results with Prim-SPH over a set of instances from the SteinLib, led to the following conclusions: the distributed DA algorithm (together with the Prim-SPH over a restricted graph) indeed provides better solutions on average and also provides tight lower bounds giving strong guarantees of the quality of those solutions; in spite of having larger worst-case complexities, the practical performance of distributed DA in terms of time and number of messages showed to be quite competitive with those of Prim-SPH; as distributed DA does not require executing a distributed shortest distance algorithm in advance, it even showed a better overall performance on instances having denser graphs.

To summarize, we believe that distributed DA can be considered as a practical alternative to the Steiner problem in graphs, specially in the context of multicast routing.

References

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

Chen, G., Houle, M., and Kuo, M. (1993). The steiner problem in distributed computing systems. *Information Sciences*, 74:73–96.

de ARAGÃO, M., Uchoa, E., and Werneck, R. (2001). Dual heuristics on the exact solution of large steiner problems. *Electronic Notes in Discrete Mathematics*, 7:46–51.

de ARAGÃO, M. and Werneck, R. (2002). On the implementation of mst-based heuristics for the steiner problem in graphs. In *ALENEX '02: Revised Papers from the 4th International Workshop on Algorithm Engineering and Experiments*, pages 1–15.

Gallager, G., Humblet, A., and Spira, M. (1983). A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Languages and Systems*, 5:66–77.

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

Goemans, M. and Williamson, D. (1996). The primal-dual method for approximation algorithms and its application to network design problems. In *Approximation algorithms for NP-hard problems*, pages 144–191. PWS Publishing Co., Boston, MA, USA.

Koch, T., Martin, A., and Voss, S. (2000). Steinlib: an updated library on steiner problems in graphs. *ZIB-Report*, pages 00–37.

Novak, R., Rugelj, J., and Kandus, G. (2001a). A note on distributed multicast routing in point-to-point networks. *Computers and Operations Research*, 28:1149–1164.

Novak, R., Rugelj, J., and Kandus, G. (2001b). Steiner tree based distributed multicast routing in networks. In *Steiner Trees in Industries*, pages 327–352. Kluwer Academic, Norwell, MA, USA.

Oliveira, C. and Pardalos, P. (2005). A survey of combinatorial optimization problems in multicast routing. *Computers and Operations Research*, 32:1953–1981.

Polzin, T. and Vahdati, S. (2001). Improved algorithms for the steiner problem in networks. *Discrete Applied Mathematics*, 112:263–300.

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

Segall, A. (1983). Distributed network protocols. *IEEE Trans. on Information Theory*, 29:23–35.

Singh, G. and Vellanki, K. (1998). A distributed protocol for constructing multicast trees. In *OPODIS '98:Distributed Computing, 2nd International Conference On Principles Of Distributed Systems*, pages 61–76.

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

Werneck, R. (2001). Steiner problem in graphs: primal and dual and and exact algorithms. Master's thesis, Catholic University of Rio de Janeiro.

Wong, R. (1984). A dual ascent approach for steiner tree problems on a directed graph. *Mathematical Programming*, 28:271–287.