A Very Fast (Linear Time) Distributed Algorithm, on General Graphs, for the Minimum-Weight Spanning Tree

Lélia Blin, Franck Butelle

A VERY FAST (LINEAR TIME) DISTRIBUTED ALGORITHM, ON GENERAL GRAPHS, FOR THE MINIMUM-WEIGHT SPANNING TREE

LELIA BLIN AND FRANCK BUTELLE

Abstract. In their pioneering paper [4], Gallager et al. introduced a distributed algorithm for constructing the minimum-weight spanning tree (MST), many authors have suggested ways to enhance their basic algorithm. Most of these improved algorithms have been shown to be very efficient in terms of reducing their worst-case communication and/or time complexity measures. In this paper, we address the problem of MST construction on general graphs. We present a new distributed algorithm for constructing the MST, which is linear in time and is faster than all previous linear algorithms. This algorithm is not to be compared with sublinear algorithms constructed on special graphs (graph diameter in order of $\log n$) [5, 6].

1. Introduction

In this paper we focus on the problem of finding a distributed algorithm for a minimum-weight spanning tree, it is a fundamental problem in the field of distributed network algorithms. Trees are an essential structure in various communication protocols, e.g Network Synchronization, Bread-First-Search and Deadlock Resolution. For the purpose of disseminating information in the network, it is advantageous to broadcast it over a minimum-weight spanning tree, since information will be delivered to every node with small communication cost.

The problem of finding a leader is reducible to the problem of finding a spanning tree. Among other systems applications, leader election is used in order to replace a malfunctioning central lock coordinator in a distributed database, for finding a primary site in a replicated distributed file system, etc.

To summarize, construction of a spanning tree or finding a leader appears as a building block essentially in every complex network protocol, and is closely related to many problems in distributed computing.

In this paper, we consider the problem of finding the Minimum-weight Spanning Tree [9]: Let $G = (V, E)$ be a weighted graph, where $w(e)$ denotes the weight of edge $e \in E$ ($n = |V|, m = |E|$). The weight of a spanning

tree T of G equals the sum of the weights of the $n-1$ edges contained in T, and T is called a *Minimal Spanning Tree*, if no tree has a smaller weight than T. It is assumed here that each edge has unique weight, i.e., different edges have different weights, and it is a well-known fact that in this case there is an unique Minimal Spanning Tree. But, having distinct weights is not an essential requirement, since one can always "create" them by appending the identities of the nodes connected by the edge, the smaller of the two first.

In a distributed algorithm, each node in V is associated with its own processor, and processors are able to communicate with each other via the edges in E, each edge is associated with a bidirectional link. The goal is to have the nodes cooperate to construct a tree covering the nodes in V whose total edge-weight is not greater than any other spanning tree for G.

In their pioneering paper [4], Gallager, Humblet and Spira introduced the distributed MST problem and presented an algorithm that has formed the basis of subsequent work in the area, for example [1, 2, 3, 5, 6]. All algorithms to compute the MST are based on the notion of a *fragment*, which is a subtree of the MST. Initially each node is a one-node-fragment. The nodes in the fragment cooperate to find the minimum-weight outgoing edge of the fragment. When this edge is known, the fragment is combined with the other fragment by adding the outgoing edge. The algorithm terminates when only one fragment remains.

These steps define the general computing approach of the MST, but the improvements are due to the different constructions inside each step. For example, [4] builds the fragments, and after, the nodes in the fragment cooperate to find the minimum-weight outgoing edge of the fragment, this cooperation is expensive in time and messages. To improve the time complexity [5] (improved by [6]) uses a "small" number of fragments, all of which have a "small" diameter. A broadcasting BFS is the best algorithm for time complexity but expensive for the number of messages exchanged.

Algorithms like [1, 2, 3, 4] search for an optimal time and message complexity, on the contrary, [5] concentrate on time complexity, and ignore the communication cost of their algorithms. When authors focus on time complexity, the message economy come second. One reason is that in "real life" the two complexity measures are not independent. However fast computing is not easy, the algorithm in [5] uses sub-linear (i.e, less than $O(n)$) time, but the solution appears a bit artificial [10].

To improve again the time complexity, the distributed construction of an MST is based on synchronous, coordinated and centralized operations in [5]. Moreover [5] suggest that the diameter of the network is a more accurate parameter for describing the time complexity, however they do not use a general

graph, but a graph with a network diameter D in order of $\log n$. For the general graph, a may be more accurate parameter for describing the time complexity is the diameter of the resulting MST.

Our asynchronous algorithm belongs to the context of the search of the best time complexity with a reasonable number of exchanged messages (in order of n^2 in the worst case, let us recall that a complete graph have almost n^2 edges). For a general graph, the distributed MST problem requires at least $O(m + n \log n)$ messages, where the transmission of one message across one edge costs one unit of time. The time complexity of [4] asynchronous algorithm is $O(n^2)$ and its message complexity is $2m + 5n \log n$. [1, 2] improve the time complexity (respectively $\simeq 4n$ and $n \log^* n$), but not the message complexity. Our algorithm propose a solution for computing the distributed MST in time $n/2$ with $O(n^2)$ messages.

2. The MST Problem

2.1. The Model

Now, we consider the standard model of static asynchronous network. This is a point-to-point communication network, described by an undirected communication graph (V, E) where the set of nodes V represents network of processors and the set of edges E represents bidirectional non-interfering communication channels operating between neighboring nodes. No common memory is shared by the nodes' processors.

All the processors have distinct identities. However each node is ignorant of the global network topology except for its own edges, and every node does know identity of its neighbors. This assumption is only used to simplify algorithm presentation, knowing neighbor's identities is not an essential requirement, since every node can send its identity to its neighbors in the first message.

We confine ourselves only to event-driven algorithms, which do not use time-outs, i.e. nodes cannot access a global clock in order to decide what to do. This is a common model for static communication networks [4].

The following complexity measures are used to evaluate performance of distributed algorithms. The Message Complexity, is the worst case total number of messages exchanged. The time complexity is the maximum possible number of time units from start to the completion of the algorithm, assuming that the inter-message delay and the propagation delay of an edge is at most one time unit of some global clock. This assumption is used only for algorithm analysis, but can not be used to prove its correctness, since the algorithm is event-driven.

2.2. The Problem

In a distributed algorithm for the Minimum Spanning Tree problem, each node has a copy of a node algorithm determining its response to every kind of message received. Namely, the algorithm specifies which computation should be done and/or which message shoud be sent. The algorithm is started independently by all nodes, perhaps at different times. At the start time, each node is ignorant of the global network topology except for its own adjacent edges. Upon the termination of the algorithm, every node knows its neighbors in the Minimum Spanning Tree.

Without loss of generality, we assume that all links are assigned distinct weights with a total ordering defined on the domain of the weights. This condition guarantees uniqueness of the MST. It is easy to achieve this, as observed in [4], by simply assigning the weight of each link (i, j) as the tuple $[\min(i, j), \max(i, j)]$ and comparing these tuples lexicographically.

3. Our Algorithm

3.1. Description of the BB01 Algorithm

The algorithm presented here has only three steps. We assume that any non-empty subset of nodes awake and start the algorithm spontaneously. The other nodes awake upon message receipt.

Initial Step

In this first step, each one-node-fragment sends information to all its neighbors. Namely, each node sends the message $< Child >$ to the node at the other endpoint of the minimum-weight edge of its adjacent edges, and sends the message $< NoChild >$ to all the other neighbors.
At the end of this step every node knows its children and its parent in its future fragment (this "direction" in the MST can change during the other steps). Moreover, each node knows if it is a leaf in the tree fragment or not. Note that a fragment can have two "roots", each one is the parent-child of the other. These "roots" will be called "Decision Centers" (or DC for short) in the following, see figure 1.

Data Step

In this second step, every leaf sends their outgoing edges and their identities to their parents. An internal node receives the informations from its children and

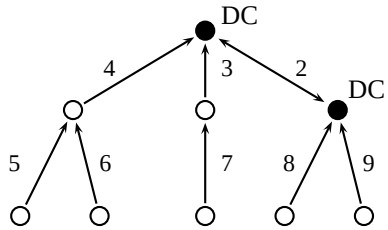

Figure 1: Data Step

forward the informations merged to the DC via its parent.
At the end of this step, the DC compute, from the information received, the minimum-weight outgoing edge, this edge becomes the merging edge with another fragment. If there are two Decision Centers, they achieve the same computation, so they both discover in which direction is the outgoing edge, and then decide automatically which one of the two DC will have to start the Connect Step.
The direction in the fragment tree is arbitrary, thus we can change it. If a node receives information from its parent, and if it misses only one child information, this child can become its parent. More clearly, the idea is to fetch information without waiting. This improvement decreases the time complexity particularly when the MST is very deep (at worst when it is a chain it divide by two the time complexity that we will obtain if we will wait instead for the child's message).

Connect step

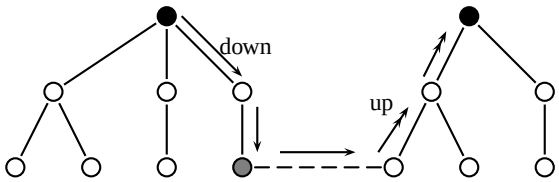

Figure 2: Connect Step

In this last step, the different fragments merge together. When a DC finds the minimum-weight outgoing edge of the fragment, it sends a $< down >$ message with the outgoing edges of the fragment toward the endpoint of this edge which is in the fragment (see figure 2). This information goes through only the branch conducting to the merging node, that bounds the message complexity.

When a node receives this message from another fragment, it sends it to its DC (message $< up >$, see figure 2), which then, computes the new minimum-weight outgoing edge.
This step is finished when the fragment is unique, namely when the fragment does not have any more outgoing edge.

3.2. Detailed Description of the BB01 Algorithm

We will consider the variables and algorithm used by some node x.

Variables

- $Wait$: integer; (* count-down the answers *)
- $Parent$: Parent of x or nil.
- $OtherCenter$: When there is two DC, and x is a DC, this variable indicates the other center identity.
- $Children$: Set of the children of x.
- $nodesfrag$: Received set of the nodes fragment.
- $OutEdges$: Set of (a, b, ω) uplet. $(a, b, \omega) \in E$, $\omega = w(a, b)$
- $Redges$: Intermediated variable used to store the Received set of $OutEdges$.
- $Rnodes$: Intermediated variable used to store the Received set of $nodesfrag$.

Now let us consider what kind of messages can be received by x :

Messages exchanged

- $< child >$: The sender becomes a child of x, and x decrease $Wait$.
- $< noChild >$: x just decrease $Wait$.
- $< data, Rnodes, Redges >$: The sender gives x its outgoing edges $Redges$. $Redges$ is the set of (a, b, ω) where a, b, ω are defined as above.
- $< down, Rnodes, Redges >$: Upon receipt of this message, the sender becomes a child of x. If this message comes from a node not belonging to the fragment, send an $< up >$ message to the parent.
- $< up, Rnodes, Redges >$: The sender is a child of x. Upon receipt of this message, if the receiver is a DC, it computes the minimum-weight outgoing edge else it sends the new information to its parent...

Subroutines

The first subroutine, **INIT**, initializes all the variables, in particular the $Wait$ variable which counts the neighbors' answers, $Wait = |Neighbors|$. Moreover, it computes the minimum-weight adjacent edge of the node. The corresponding endpoint becomes its parent. The node sends the $< child >$ message to its parent and the $< noChild >$ message to all others neighbors.

The next subroutine is **NEXSTEP**. In this subroutine the node has received $< Child >$ or $< noChild >$ messages from all its neighbors. From now the $Wait$ variable count the number of received children's messages. A node who has no child or a node who has only one child which is also its parent is a leaf (it is a DC but without interest). A leaf sends $< data, Rnodes, Redges >$ message to its parent.

EdgesManagement subroutine merges the received set with its $nodesfrag$ set and its $OutEdges$ set. If needed, it deletes the cycles.

In **CHANGEROOT** subroutine, x has received data messages from its parent and from all of its children minus one. This missing node becomes the new parent of x. The node x send its information to its new parent.

In **CHOOSE** subroutine, x computes the minimum-weight outgoing edge of the fragment (it is a DC). It sends the $< down >$ message, either to the other endpoint of this adjacent edge, or to a fragment node which drive to this edge. In this second case, x looses the DC property.

The keyword **END** is not a subroutine, it means only the end of the algorithm. If a process termination is needed, this must be changed into a subroutine broadcasting an end message via the MST.

Detailed Algorithm

1. Spontaneous awakening (or upon message receipt)
   - ▷ **INIT**
2. Upon receipt of $< child >$ from y
   - ▷ $Wait - -$
   - ▷ if $parent \neq y$ then $children := children \cup \{y\}$
   - ▷ else $OtherCenter := y$
   - ▷ $OutEdges = OutEdges - \{(x, y)\}$
   - ▷ if $Wait = 0$ then **NEXTSTEP**
3. Upon receipt of $< noChild >$ from y
   - ▷ $Wait - -$
   - ▷ if $Wait = 0$ then **NEXTSTEP**
4. Upon receipt of $< data, Rnodes, Redges >$ from y
   - ▷ $children := children \cup \{y\}$
   - ▷ $Wait - -$

- ▷ **EdgesManagement**
- ▷ if $|OutEdges| = 0$ then *END*
- ▷ if $(Wait = 0)$ then
  - □ if $(parent = nil \wedge parent \neq y)$ then
    send$< data, nodesfrag, OutEdges >$ to $parent$
  - □ else
    - ◇ if $(parent = y)$ then
      $OtherCenter := y$
      $parent := nil$
    - ◇ **CHOOSE**
- ▷ else if $(parent = y)$ then
  - ◇ $OtherCenter := nil$
  - ◇ $parent := nil$
- ▷ if $(Wait = 1)$ then
  - ◇ **CHANGEROOT**

5. Upon receipt of $< down, Rnodes, Redges >$ from y
   - ▷ if $(Wait = 0)$ then
     - □ $children := children \cup \{y\}$
     - □ **EdgesManagement**
   - ▷ else
     - □ if $(nodesfrag \subset Rnodes)$ then
       - ◇ $nodesfrag := Rnodes$
       - ◇ $OutEdges := Redges$
       - ◇ $children := children \cup \{y\}$
       - ◇ $parent = nil$
     - □ else **EdgesManagement**
   - ▷ if $|OutEdges| > 0$ then
     - □ if $(parent \neq nil \wedge parent \neq y)$ then
       send $< up, Rnodes, Redges >$ to $parent$
     - □ else
       - ◇ if $(parent = y)$ then
         $Othercenter := y$
         $parent := nil$
       - ◇ **CHOOSE**
   - ▷ else *END*
6. Upon receipt of $< up, Rnodes, Redges >$ from y
   - ▷ **EdgesManagement**
   - ▷ if $|OutEdges| = 0$ then *END*
   - ▷ else
     - □ if $(parent = y)$ then
       $Othercenter := y$
       $parent := nil$
       CHOOSE
     - □ else send $< up, Rnodes, Redges >$ to $parent$

3.3. Correctness

Basic properties of MST

The following properties are crucial for the correctness proofs. See for example [4] for detailed proofs.

1. If all edge weights of a connected graph G are distinct, then G has exactly one MST.
2. Let us assume that all edge weigths are distinct. Let F be a fragment of the MST and if e is the minimum-weight outgoing edge of F, then edge e can be added to F, to create a new fragment without creating a cycle. This new fragment is still a fragment of the MST.

Corollary 1. *When, (with the same assumptions) for a fragment F, there is no more outgoing edge, the construction is terminated and F is the MST.*

4. Complexity

4.1. A worst case example

When the graph is a complete graph, and its MST a chain, it is clearly a bad case for every algorithm that uses the edges of the MST to transport information, updates and so on.

The figure 3 describes what appends for our algorithm on such a case, after the **INIT** step. To be a very bad case, edges-weights are distributed alternatively on the MST chain.

4.2. Communication complexity

Theorem 2. *The total number of messages in the worst case on a graph $G = (V, E)$ is $2m + \frac{n^2}{8} + O(n)$ where $n = |V|$ and $m = |E|$.*

Proof. Let us describe the exchanged number of messages during each step (we will assume that, for sake of simplicity, all nodes awake spontaneously).
First the initial messages, when the nodes send $< Child >$ and $< noChild >$. In this step every node sends a message to all its neighbors, thus the two endpoints of an edge send a message. For this step the communication cost is clearly $2m$. Note that the number of fragments created by the first step is at most $\lfloor \frac{n}{2} \rfloor$, each fragment is composed at least of 2 nodes (can be 3 if n is odd).
Second the Data Step. The worst case appends when the MST is a chain and the number of fragments is maximal (see the column d of figure 3). In this case, every node in each fragment sends its information to the other fragment

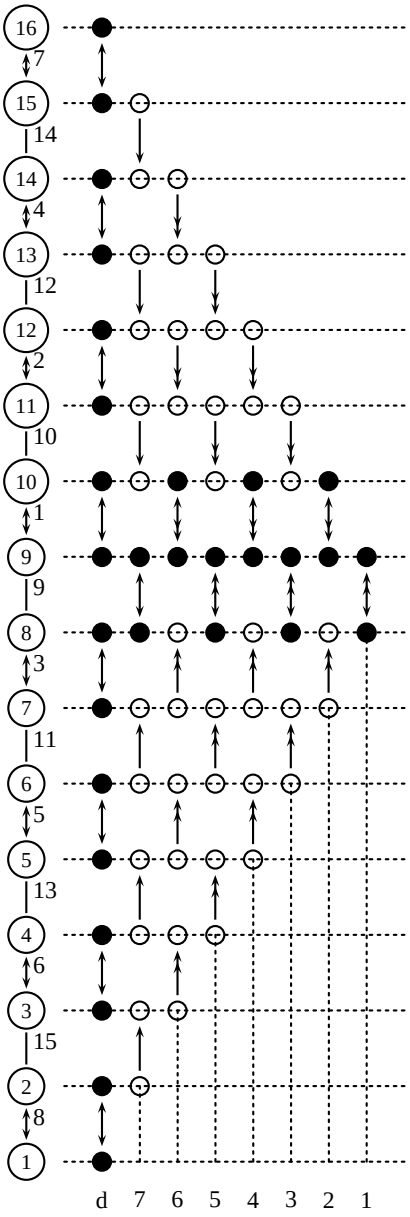

Figure 3: A worst case example

node and that is all for the step. Note that the number of data messages can be greater with bigger fragments, but there is a tradeoff between the number of fragments and their size. In this data step there is n messages exchanged.
Finally, the connect step is represented by all the other columns in the figure 3. We will consider levels. A level will be defined by the number of fragments. We go from level $i+1$ to level i when the number of fragments is divided by 2 (to simplify the analysis we will suppose without loss of generality that $n = 2^k$). To go from level $i+1$ to i, the $< down >$ messages have to cross its fragment and next go through its best outgoing edge. The $< up >$ messages have to cross the fragment to "chase" the DC. From construction and since we are interested in the worst case, one fragment will absorb all the others, leading to a great number of $< up >$ messages plus an oscillation of the $< down >$ messages as shown on the figure. We define level 1 when there is only one remaining fragment. So we start at level $n/2 - 1$ with at most $n/2$ fragments of size 2. It gives for these last steps $\Sigma_{i=1}^{n/2-1}(i+1) = \frac{n^2}{8} + \frac{n}{4} - 1$. All in all the sum defined in the theorem holds.
If the growth of the fragments is equilibrate, see figure 4.2, the exchanged messages are mostly $< down >$ messages and the sum is in order of $2m + \frac{1}{2}n\log(n) + O(n)$. □

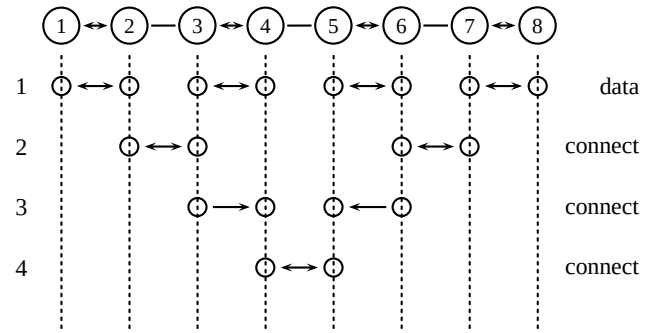

Figure 4: An example

4.3. Time complexity

Theorem 3. *If all nodes awake spontaneously almost at the same time, the worst case time complexity of this algorithm is* $\frac{n}{2} + O(1)$.

Proof. The worst case is when the MST is a chain over a complete graph (see figures 3 and 4.2). So the time complexity follows from the previous proof, the messages have to cross the chain from both endpoints. When they arrive at the middle, all information have been collected and the algorithm is finished (no more outgoing edge). □

References

[1] B. Awerbuch *Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems*, ACM Symposium on Theory Of Computing pp. 230–240, 1987.

[2] F. Chin and H.F. Ting *An almost linear time and $O(nlogn+e)$ messages distributed algorithm for minimum-weight spanning trees* Proc. of the IEEE Sym. on FOCS, 1985.

[3] M. Faloutsos and M. Molle, *Optimal distributed algorithm for minimum spanning trees revisited*, Fourteenth ACM Symposium on Principles of Distributed Computing (PODC'95), Ottawa, Ontario, August 1995.

[4] R.G. Gallager, P.A. Humblet and P.M. Spira *A Distributed Algorithm for Minimum Weight Spanning Trees*, ACM Trans. On Programming Languages Systems, Vol. 5, pp. 66–77, January 1983.

[5] J.A. Garay, S. Kutten and D. Peleg *A Sublinear Time Complexity Distributed Algorithm for Minimum-Weight Spanning Trees* SIAM J. COMPT. Vol 27, pp 302-316, February 1998.

[6] S. Kutten and D. Peleg *Fast distributed construction of small k-dominating sets and applications* J. Alg. Vol 28, pp 40-66, 1998.

[7] I. LavallÈe, G. Roucairol, *A fully distributed (minimal) spanning tree algorithm* Information Processing Letters, Vol. 23, pp. 55–62, 1986.

[8] D. Peleg and V. Rubinovich *A Near-Tight Lower Bound on the Time Complexity of Distributed Minimum-Weight Spanning Tree Construction* SIAM J. COMPT. Vol 30, No. 5, pp-1427-1442, 2000.

[9] G. Tel *Introduction to Distributed Algorithms* Cambridge University Press, 1994.

[10] G. Tel *Distributed Control for AI* Distributed Artificial Intelligence, Ch. 14, MIT Press, G. Weiss ED., 1998.

Authors addresses:
L. Blin
L.R.I.A, Univerity of Paris 8
15 Rue Catulienne, 93200 Saint-Denis, FRANCE
lelia.blin@univ-paris8.fr
F. Butelle
L.I.P.N, University of Paris 13, UPRES-A 7030
franck.butelle@lipn.univ-paris13.fr