Construction auto-stabilisante d'un arbre couvrant de poids minimum

Lélia Blin, Shlomi Dolev, Maria Potop-Butucaru, Stephane Rovedakis

Construction auto-stabilisante d'un arbre couvrant de poids minimum†

L. Blin[1] and S. Dolev[2] and M. Potop-Butucaru[3] and S. Rovedakis[4]

[1]*Université d'Evry-Val d'Essonne, 91000 Evry, France. LIP6-CNRS UMR 7606, France.*
[2]*Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel.*
[3]*Université Pierre & Marie Curie - Paris 6, 75005 Paris, France. LIP6-CNRS UMR 7606, INRIA REGAL, France.*
[4]*Laboratoire CEDRIC-EA1395 - CNAM, 292 Rue St Martin, 75141 Paris Cedex 03, France.*

L'arbre couvrant de poids minimum offre une solution de routage ayant le double avantage de donner une structure de communication simple et économique. Dans ce papier, nous présentons un nouvel algorithme auto-stabilisant pour la construction d'un arbre couvrant de poids minimum dans un système distribué et asynchrone. Notre solution améliore l'existant en permettant d'atteindre un meilleur compromis entre le temps de convergence, $O(n^2)$, et la complexité en mémoire nécessaire sur chaque noeud du réseau, $O(\log^2 n)$. Le temps de convergence est amélioré d'un facteur multiplicatif $\Theta(n)$ au prix d'un facteur multiplicatif de $O(\log n)$ sur la mémoire. La clé de voûte de ce travail est l'utilisation d'une méthode de nommage auto-stabilisante permettant d'identifier pour toute paire de noeuds le plus proche ancêtre commun dans l'arbre.

Keywords: Algorithme distribué, Auto-stabilisation, Construction d'arbre couvrant de poids minimum.

1 Introduction

Since its introduction in a centralized context, the minimum spanning tree (or MST) problem gained a benchmark status in distributed computing thanks to the seminal work of Gallager, Humblet and Spira [GHS83].

The emergence of large scale and dynamic systems, often subject to transient faults, revives the study of scalable and self-stabilizing algorithms. A *scalable* algorithm does not rely on any global parameter of the system (*e.g.* upper bound on the number of nodes or the diameter). *Self-stabilization* introduced first by Dijkstra in [Dij74] deals with the ability of a system to recover from catastrophic situation (i.e., the global state may be arbitrarily far from a legal state) without external (*e.g.*, human) intervention in finite time.

Although there already exists self-stabilizing solutions for the MST construction, none of them considered the extension of the Gallager, Humblet and Spira algorithm (GHS) to self-stabilizing settings. Interestingly, this algorithm unifies the best properties for designing large scale MSTs : it is fast and totally decentralized and it does not rely on any global parameter of the system. Our work proposes an extension of this algorithm to self-stabilizing settings. Our extension uses only logarithmic memory and preserves all the good characteristics of the original solution in terms of convergence time and scalability.

Related Works. Gupta and Srimani presented in [GS03] the first self-stabilizing algorithm for the MST problem. This MST construction is based on the computation of all shortest paths (for a certain cost function) between all pairs of nodes. The time complexity announced by the authors, $O(n)$ stays only in the particular synchronous settings considered by the authors, in asynchronous setting the complexity is $\Omega(n^2)$ rounds. A different approach was proposed by Higham and Liang [HL01]. The algorithm checks every edge whether it eventually belongs to the MST or not. The memory used by each node is $O(\log n)$ bits, but the information exchanged between neighboring nodes is of size $O(n \log n)$ bits, thus only slightly improving that of [GS03]. Its computation is expensive in large scale systems. The time complexity of this approach is $O(mD)$ rounds where m and D are the number of edges and the diameter of the network respectively, i.e., $O(n^3)$ rounds in the worst case.

†La version longue de ce papier a été publié à la conférence DISC 2010 et est supporté par le projet *ANR SHAMAN*.

	a priori knowledge	space complexity	convergence time
[GS03]	network size and the nodes in the network	$O(n\log n)$	$\Omega(n^2)$
[HL01]	upper bound on diameter	$O(\log n)$ +messages of size $O(n\log n)$	$O(n^3)$
[BPBRT09]	**none**	$\mathbf{O(\log n)}$	$O(n^3)$
This paper and [BDPBR10]	**none**	$O(\log^2 n)$	$\mathbf{O(n^2)}$

TAB. 1: Distributed Self-Stabilizing algorithms for the MST problem

The main drawback of these solutions is its lack of scalability since each node has to know and maintain global parameter of the system (network size, diameter, ...). In [BPBRT09] we proposed a self-stabilizing loop-free algorithm for the MST problem. The proposed solution improves on the memory space usage since each participant needs only $O(\log n)$ bits while preserving the same time complexity as the algorithm in [HL01].

Contributions. Our contribution published also in [BDPBR10] is therefore twofold. We propose for the first time in self-stabilizing settings a $O(log^2 n)$ bits scheme for computing the nearest common ancestor. Furthermore, based on this scheme, we describe a new self-stabilizing algorithm for the MST problem. An exhaustive state of art is proposed in Table 1. Our algorithm does not make any assumption on the network size (including upper bounds) or the existence of an a priori known root. Moreover, to our knowledge our solution has the best existing space/time compromise over the existing self-stabilizing MST solutions. The convergence time is $O(n^2)$ asynchronous rounds and the memory space per node is $O(\log^2 n)$ bits.

2 Model and notations

We consider an undirected weighted connected network $G = (V,E,w)$ where V is the set of nodes, E is the set of edges and $w : E \to \mathbb{N}$ is a positive cost function. We note n the number of nodes in the network (i.e., $|V| = n$). Nodes represent processors and edges represent bidirectional communication links. Each node v has a unique identifier noted Id_v. The processors asynchronously execute their programs consisting of a set of local variables and a finite set of rules.

In the sequel we consider the system can start in any configuration. That is, the local state of a node (the value of the local variables of the node and the state of its program counter) can be corrupted. We don't make any assumption on the bound of corrupted nodes. In the worst case all the nodes in the system may start in a corrupted configuration. In order to tackle these faults we use self-stabilization techniques [Dol00].

3 Our self-stabilizing solution to MST problem

The central notion in the GHS approach [GHS83] is the notion of *fragment*. A fragment is a partial spanning tree of the graph, i.e., a fragment is a tree which spans a subset of nodes. An *outgoing edge* of a fragment F is an edge with a unique endpoint in F. The *minimum-weight outgoing edge* of a fragment F is the outgoing edge of F with minimum weight, denoted in the following as ME_F. In the GHS construction, initially each node is a fragment. For each fragment F, the GHS algorithm identifies the ME_F and merges the two fragments endpoints of ME_F. The merging process is recursively repeated until a single fragment remains, which is a MST.

Our solution combines both the blue and red rules [ST83]. The blue rule application needs that each node identifies its own fragment. The red rule requires that nodes identify the fundamental cycle corresponding to every adjacent non-tree-edge. In both cases, we use a self-stabilizing labeling scheme (see Section 3.1) which provides at each node a distinct informative label such that the nearest common ancestor of two nodes can be identified based only on the labels of these nodes. Thus, the advantage of this labeling is twofold. First the labeling helps nodes to identify their fragments. Second, given any non-tree edge $e = (u,v)$, the path in the tree going from u to the nearest common ancestor of u and v, then from there to v, and finally back to u by traversing e, constitute the *fundamental cycle* C_e.

3.1 Self-stabilizing Nearest Common Ancestor Labeling scheme

Our labeling scheme uses the notions of *heavy* and *light* edges introduced in [HT84]. In a tree, a *heavy* edge is an edge between a node *u* and its largest subtree. The other edges between *u* and its other children are tagged as *light* edges. We extend this edge designation to the nodes, a node *v* is called *heavy node* if the edge with its parent is a heavy edge, otherwise *v* is called *light node*. Moreover, the root of a tree is a heavy node. The idea of the scheme is as follows. A tree is recursively divided into disjoint paths : the *heavy paths* and the *light paths* which contain only heavy and light edges respectively.

To label the nodes in a tree T, the size of each subtree is needed to identify heavy edges at each level of T. To this end, each node *v* maintains a variable named $size_v$ which is a pair of integers : (i) a local estimation of the size of the subtree rooted at *v*, and (ii) the identifier of *v*'s child in the largest subtree. These information are used by *v* to identify the heavy edge leading to a child. The computation of $size_v$ is processed in a bottom-up fashion. A leaf node *v* has no child, therefore $size_v = (1, \perp)$. Moreover, each node *v* in T maintains a pointer towards its parent stored in variable p_v.

Based on the heavy and light nodes in a tree T (using locally variable $size_v$), each node *v* can compute its label in a top-down fashion. The label of a node *v* stored in variable ℓ_v is a list of pair of integers. Each pair of the list contains : (i) the identifier of a node, and (ii) the distance to the root of the heavy path. The root *v* of a fragment has a label equal to $(\mathsf{Id}_v, 0)$, respectively the identifier of *v* and the distance to itself, otherwise it corrects its label. When a node *u* is tagged by its parent as a heavy node (i.e., $size_{p_v}[1] = \mathsf{Id}_u$), then *u* takes the label of its parent but it increases by one the distance of the last pair of the parent label. For example, if the label of *u*'s parent is $(3,1)(2,0)$ then *u*'s label is equal to $(3,1)(2,1)$. When a node *u* is tagged by its parent *v* as a light node (i.e., $size_{p_v}[1] \neq \mathsf{Id}_u$), then *u* becomes the root of a heavy path and it takes the label of its parent to which it adds a pair of integers composed of its identifier and a zero distance. For example, if the label of *u*'s parent is $(3,1)(2,0)$ and *u*'s identifier is 14 then *u*'s label is equal to $(3,1)(2,0)(14,0)$. A heavy or light node with a locally incorrect label can correct its label using the one of its parent in the tree.

The labels given by this scheme to the nodes of a tree can be ordered following a lexicographic order. Given two nodes u, v in a tree T, the label of the *nearest common ancestor* of *u* and *v* is obtained by taking the smallest (following the lexicographic order) common part (considering node identifiers) between ℓ_u and ℓ_v. For example, if the label of *u* (resp. *v*) is $(3,2)(13,0)$ (resp. $(3,1)(2,1)$) then the common part is $(3,2)$ and $(3,1)$ which gives $(3,1)$ for the nearest common ancestor label for *u* and *v*.

3.2 Self-stabilizing algorithm for the minimum spanning tree problem

In this section we describe our self-stabilizing algorithm for constructing the minimum spanning tree. our algorithm executes two phases : the first phase corrects the existing structure (Tree, cycle) and the second phase merges the fragments. Our algorithm uses the GHS approach to merge the fragments using minimum outgoing edges to construct a spanning tree, and deletes the edge of maximum weight in each fundamental cycle to recover from invalid configurations. In both cases, it uses the labeling algorithm to identify fragments and fundamental cycles. In the following, *merging* operations have a higher priority than the *recovering* operations. That is, the system recovers from an invalid configuration iff no merging operation is possible.

The minimum weighted edge. According to the labeling scheme described in Section 3.1, given a (correct) fragment F any node $v \in F$ is able to locally detect among its neighbors the ones which belong to F by comparing the labels. To this end, *v* computes the nearest common ancestor with its neighbors. If *v* has no common ancestor with a neighbor *u* (i.e., the labels are totally distinct) then *u* belongs to a fragment different than *v*'s fragment, otherwise *u* and *v* are in the same fragment.

Each fragment computes its minimum outgoing edge in a bottom-up fashion. To this end, each node sends to its parent the outgoing edge of minimum weight among its adjacent edges and edges sent by children. To store the information related to an outgoing edge $e = (x,y)$, each node *v* maintains a variable mwe_v which is a pair of values : (i) the weight of edge *e*, and (ii) the label of the common ancestor of *x* and *y*. When the root of the fragment has computed its outgoing edge *e* of minimum weight (i.e., the minimum outgoing edge of the fragment), then it can start a merging operation (described in Paragraph "Fragments merging"). The edge *e* belongs to a MST and can be used to perform a merging between the two adjacent fragments.

When a node v in fragment F has no adjacent outgoing edge or outgoing edge sent by a child, then v sends to its parent information concerning *internal edges* of F (edges not in F whose extremity nodes are in F). These edges are sent in a bottom-up fashion in F following fundamental cycles. These information are used to repair the fragment if necessary (described in Paragraph "MST correction").

Fragments merging. When the minimum outgoing edge $e = (u,v)$ of a fragment F_v is computed by the root r of F_v, then a merging operation is started by r. To this end, the parent pointers are reversed on the path between r and v in F_v (i.e., following the nodes x in F_v such that $mwe_x = mwe_r$). During this reorientation the labels are locked. That is, each node x on the path between r and v (including r and excluding v) changes its label to : $\ell_x := (\perp, \perp)$. When v becomes the root of the fragment F_v it can merge with the fragment F_u. Note that, the node among u and v with the smallest identifier becomes the root of the new fragment F given by the merging of F_u and F_v. After the addition of the outgoing edge e, the labeling process is re-started in F. The merging phase is repeated until a single fragment is obtained (i.e., there is no outgoing edges of a fragment).

MST correction. The system can start in any arbitrary configuration, so we must be able to detect incorrect fragments (i.e., with an edge part of no MST). Therefore, a node v with an incoherent parent (which is not in its neighborhood) or present in a cycle (its label is contained or is inferior to its parent label) becomes the root of a fragment by setting its parent pointer to void and its label to $(\mathsf{Id}_v, 0)$.

Consider a part of a fragment with no outgoing edge. In this case, each node v piggy back up information concerning internal edges (stored in variable mwe_v). The edges are sent following an order on the label of the nearest common ancestor associated to edges (the ancestor nearest to the root of fragment first). The information concerning an internal edge e is sent following the fundamental cycle of e and its transmission is stopped at the nearest common ancestor associated to e. If the parent x of a node v has the same information with v about an internal edge e and the weight of the edge $w(v,x) > w(e)$ then the edge (v,x) is removed from the tree, since (v,x) can be the edge of maximum weight in the fundamental cycle C_e.

Theorem 1 *Starting from any arbitrary configuration our algorithm eventually constructs a minimum spanning tree in at most $O(n^2)$ rounds using $O(\log^2 n)$ bits of memory per node.*

Références

[BDPBR10] Lélia Blin, Shlomi Dolev, Maria Gradinariu Potop-Butucaru, and Stephane Rovedakis. Fast self-stabilizing minimum spanning tree construction - using compact nearest common ancestor labeling scheme. In *DISC*, pages 480–494, 2010.

[BPBRT09] Lélia Blin, Maria Potop-Butucaru, Stephane Rovedakis, and Sébastien Tixeuil. A new self-stabilizing minimum spanning tree construction with loop-free property. In *DISC*, pages 407–422, 2009.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11) :643–644, 1974.

[Dol00] S. Dolev. *Self-Stabilization*. MIT Press, March 2000.

[GHS83] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1) :66–77, 1983.

[GS03] Sandeep K. S. Gupta and Pradip K. Srimani. Self-stabilizing multicast protocols for ad hoc networks. *J. Parallel Distrib. Comput.*, 63(1) :87–96, 2003.

[HL01] Lisa Higham and Zhiying Liang. Self-stabilizing minimum spanning tree construction on message-passing networks. In *DISC*, pages 194–208, 2001.

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2) :338–355, 1984.

[ST83] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3) :362–391, 1983.