Managing clusters among Distributed Dynamic Environments

Thibault Bernard
CRESTIC, URCA, BP 1039
F-51687 Reims Cedex 2
France
Email: thibault.bernard@univ-reims.fr

Alain Bui
PRIISM, UVSQ
45, avenue des États-Unis
F-78035 Versailles Cedex
France
alain.bui@prism.uvsq.fr

Laurence Pilard
PRIISM, UVSQ
45, avenue des États-Unis
F-78035 Versailles Cedex
France
laurence.pilard@prism.uvsq.fr

Abstract—We propose a fully decentralized algorithm that constructs and maintains clusters over a network. This algorithm maintains a stable size within the clusters among a network subject to frequent connection and disconnection. We use the notion of a circulating token that collects data (called a circulating word). This token moves according to a random walk scheme. The aim of the algorithm is to adapt solutions that use random walks and circulating words to large scale networks.

I. INTRODUCTION

Distributed algorithms deal with some entities geographically distributed on different sites and interconnected in a communication network. These entities work together to reach a global goal. The main advantage of a distributed solution is to share the computing resources and to increase efficiency and service availabilities. The development of new technologies as wireless communication and broadband internet access increase the utilization of distributed solutions in lots of applications.

The development of dynamic networks highlight a major problem: the reliability of communication between two arbitrary nodes in the network. In this paper, we address the problem of structuring a network using a random walk scheme. In graph theory, random walk studies [AKL+79], [Lov93] gave some very efficient results in a lot of classical problems. We already proposed some random walk based algorithms [BBF04a], [BBFR06] and studied their efficiency [BS07].

Random walks are naturally adaptive to dynamic networks as ad-hoc sensors network [BBCD02], [DSW06] because they use only a local up-to-date information. Moreover they can easily manage connections and disconnections occurring in the network.

A random walk based algorithm is a token circulation algorithm where a token randomly moves among all nodes in the network. A random walk can be used as a base of distributed token circulation algorithm. This token circulation can collect and disseminate information in the network. At each step of the execution of the algorithm, the random walk (the token) is on a node \( i \) of the network. The node that owns the token chooses one of its neighbour \( j \) with a probability \( 1/\text{degree}(i) \) (and more generally in the case of weighted graphs \( \omega(i,j)/\omega(i) \)) where \( \omega(i,j) \) is the weight of the edge \( (i,j) \) and \( \omega(i) \) is the sum of weights of all edges adjacent to \( i \). It is important to remark that this definition ensures all nodes, with high probability, will eventually own the token, and the token, with high probability, will eventually hit all nodes [Lov93].

In [BBF04a], [BBFR06], we introduced and used the combination of a circulating word, i.e. the token has a content to collect and broadcast data (this concept is formally defined in Section II-C) and a random walk as moving scheme of the circulating word. Using this combination, we proposed solutions to build adaptive spanning trees for dynamic systems as ad-hoc sensor networks (the token perpetually moves in the network, collecting data about all topological modifications in order to update the spanning tree). These solutions are tolerant to transient failure that could occur in the network. In these works, we also proposed a fully decentralized solution to the communication deadlock problem, introducing a new control mechanism called reloading wave. These works have been used to propose a solution to the resource allocation and mutual exclusion problems in networks subject to node volatility [BBF04a]. Such a combination has also been used in [BBFR06] to build and maintain spanning structures to solve on-the-fly research resources in peer-to-peer grids.

Although the token perpetually circulates in the network in order to update underlying structures, we succeed to bound the size of the circulating word to \( 2n - 1 \) in the case of bidirectional communication links and to \( n^2/4 \) in the case of unidirectional communication links, where \( n \) is an upper bound on the size of the network. The cover time \( C \), i.e. the average time to visit all nodes in the network, and the hitting time \( h_{i,j} \), i.e. the average time to hit for the first time a given node \( j \) starting from a node \( i \), are two important values in the analysis of random walks based algorithms. The cover time for an arbitrary topology is bounded in the worst case to \( O(n^3) \) [Lov93] but could be more precisely computed depending on the considered topology [BS07]. Random walks based solutions are an alternative to flooding solutions: even if they are more time consuming, they use less bandwidth than flooding. All these bounds on the size of the word or on the cover time are perfectly reasonable for limited size networks but they are not anymore for large scale networks. We address
in this work the scalability of such solutions.

We propose to use the content of the circulating word to structure the network into different clusters. The construction and the maintenance of them are achieved in a decentralized way. Using the properties of random walks and circulating word, the clusters are able to adapt to topological reconfiguration. Thus this solution can be used to design distributed control algorithm on large scale dynamic networks.

Unlike solutions described in [Bas99], [JN06], our solution does not use any local leader on a cluster. The advantage of such kind of solutions is if a node «moves» in the network, this does not necessarily implies a total reconstruction of the cluster. This minimize the number of structure modifications. However the structure maintenance requires a longer delay since it is not achieved by a local leader. Since our solution uses random mechanism to adapt the different clusters, a topological change can involve an inconsistent global state during a short period. Nevertheless the system will eventually converge to a correct global state without having to rebuilt the whole clusters. This kind of approach on a 1-hop solution is described in [TIMF05] where re-clustering mechanism are used. Finally, our solution is totally decentralized. Contrary to [4], where a spanning structure of the whole network is built in a first step, then it is divided using a global mechanism, our solution is realized in a fully concurrent way. As stated in [2], it considerably accelerates the construction of the different clusters. Thus our solution satisfies the property highlighted in [TV08].

II. PRELIMINARIES

In this section, we are going to define a distributed system, a random walk and a circulating word. We will also introduce the clustering notion.

A. Distributed system

A distributed system is a connected graph $G = (V, E)$, where $V$ is a set of nodes with $|V| = n$ and $E$ is the set of directional (bidirectional) communication links. (We use the terms «node», «vertex», «site» and «processor» interchangeably). A node is a computing unit and a message queue. A communication link $(i, j)$ exists if and only if $i$ is a neighbour of $j$. Every node $i$ can distinguish all its communication links and maintains a set of neighbours (denoted by $N_i$). The degree of $i$ is the number of neighbours of $i$, i.e. $|N_i|$ (denoted by $\text{deg}(i)$). We consider a distributed system where all nodes have distinct identities.

B. Random walk

A random walk is a sequence of vertices visited by a token that starts at $i$ and visits other vertices according to the following transition rule: if the token is at $i$ at step $t$ then at step $t+1$, it will be at one of the neighbours of $i$, this neighbour is chosen uniformly at random among all of them [Lov93], [AKL+79]. As with deterministic distributed algorithms, the time complexity of a random walk based on token circulation algorithms can be viewed as the number of steps it takes for the algorithm to achieve the network traversal.

Both the cover time $C$ — the average time to visit all nodes in the system — and the hitting time denoted by $h(i, j)$ — the average time to reach a node $j$ for the first time starting from a given node $i$ — are important values that appear in the analysis of random walk-based distributed algorithms. Original method to efficiently compute hitting times and cover time is described in [BBBS03], [BS07].

C. Circulating word

A circulating word is a list of data collected during the circulation of the word in the network. Such a word can collect different kind of data but we are interested here in collecting identifiers of visited nodes. The word is represented as follow: $w = < w[1], \ldots, w[k] >$ where $k$ is the size of the circulating word and $w[z]$ is the node identifier placed at $z^{th}$ position in the word. The word is updated as follow: each time it circulates and visits a node, the node identifier is added at the beginning of the word, i.e. at position 1.

The word circulates following a random walk scheme, allowing efficient management of dynamic network. The word perpetually circulates, collecting identifiers of all visited nodes in order to maintain an adaptive spanning tree of the network. Note that only the most recent data are used to build this spanning tree, and thus only a size of the word bounded by $2n - 1$ is needed. The detailed procedure to reduce the size of the word is described in [Ber06]. A node that owns the circulating word can compute a spanning tree over the network using algorithm described in [BBF04b].

D. Clusters

We propose to reduce the cost of application based on the combination of random walk and circulating word, by creating partitions over the network. The management of the circulating word should be adapted: one circulating word is assigned to the maintenance of each partition, and collects identities of nodes of its assigned partition. We bound the size of a partition between two values: the lower bound $m$ and the upper bound $M$. This reduces the delay to update information contained in the circulating word. In our solution, it is required that $m < M/2$.

Definition 1 (Cluster): A cluster $p$ is a set of node identified by an unique identifier noted $id_p$. The set of nodes in $p$ is noted $V_p$ and the neighborhood of the cluster $N_p = \{ i \in V \setminus V_p : \exists j \in V_p, (j, i) \in E \}$. The mobility of the nodes can imply two kinds of situations: a cluster of size strictly greater than $M$ or a cluster of size strictly lower than $m$. The aim of our algorithm is to maintain each node of the system in a stable cluster i.e. a cluster of size between $m$ and $M$.

Definition 2 (Stable cluster): A cluster $p$ is stable iff $m \leq |V_p| \leq M$.

Definition 3 (Token of a cluster): In each cluster $p$, a message randomly circulates in $V_p \cup N_p$; it is called a token. A token is associated to its cluster $p$. 
Each node in the networks either belong to a cluster or is free.

Definition 4 (Free node): A free node is a node that does not belong to any cluster. If a node is free then its cluster identifier is \( \text{null} \), otherwise, its cluster identifier is the token identifier of its cluster.

III. ALGORITHM DESCRIPTION

Each token perpetually circulates according to a random walk scheme, building and maintaining its own cluster. The aim is to build and maintain stable clusters taking into account the dynamicity of the network. All of this takes place through different phases. At the beginning, a cluster (in fact its associated token) sequentially annexes nodes, i.e. when a token meets a free node, the free node joins the cluster; it is the collect mechanism. When a token meets another cluster (maintained by another token), two cases can possibly occur: either the token returns to its own cluster or the token triggers a dissolution mechanism on its own cluster if this cluster has a size below \( m \). This mechanism implies all nodes in the cluster becomes free. The third mechanism is the division mechanism: if a cluster grows up to a size \( M \) then the cluster is divided into two smaller clusters, such that each smaller cluster becomes stable. Because our algorithm is adaptive to nodes and links volatility, the obtained result is each node eventually belongs to a stable cluster.

A. Nodes and tokens variables

A cluster is identified by a color and a version number. Each node saves the identifier of the cluster it belongs to and each token contains the identifier of the cluster it represents. When a node creates a new token, if the node is free then the node creates a token for a new cluster: the token color is then the node identifier and the token version number is 0. Otherwise the node creates a token representing the cluster the node already belongs to: the token color is the color of the node’s cluster and the token version is the version of the node’s cluster +1.

The variables of a token \( t \) are:

- \( \text{col}_t \): color of the cluster represented by \( t \)
- \( \text{vers}_t \): version number of the cluster represented by \( t \)
- \( w_t \): circulating word contained in \( t \)
- \( \text{colSender}_t \): color of the last node who sent \( t \)
- \( \text{hops}_t \): number of time \( t \) has been sent since its creation (the first sending, i.e. the one associated to its creation, is not included).

The local variables of a node \( i \) are:

- \( i \): node identifier
- \( \text{col}_i \): color of the cluster of the node \( i \)
- \( \text{vers}_i \): version number of the cluster of the node \( i \)
- \( w_i \): circulating word of the last valid token having visited \( i \)
- \( \text{timer}_i \): timer initialized to \( T_{MAX} \), when its value is 0, \( i \) creates a new token.

A free node is such that \( \text{col}_i = \text{null} \).

When a node receives a token, the node checks if the token is valid.

Definition 5 (Valid token): A token \( T \) is valid for a node \( i \) if \( \text{col}_i = \text{col}_T \land \text{vers}_i \leq \text{vers}_T \)

B. Token creation

The token creation is managed using the variables \( \text{timer}_i \) for each node \( i \) and \( \text{hops}_t \) for each token \( T \). We use the mechanism describes in [BBF04a]. The goal of this mechanism is to ensure that a token represented a cluster \( c \) is created if there is no token in \( c \). When the timer of a node is 0, this node generates a new token. When the \( \text{hops} \) value of a token reaches \( T_{MAX} \), the token initiates the propagation of a reloading wave in its cluster in order to reset to \( T_{MAX} \) the timer of all node in the cluster. In order to propagate the reloading wave, we maintain a dynamic self-stabilizing tree through the system [BBF04b]. This tree is built on-the-fly using the circulating word contained in the token.

C. Only one guarded rule

Algorithm 1 On node \( i \) awakening

1: \( \text{timer}_i \leftarrow \text{timer}_i - 1 \)
2: if \( (\text{col}_i \neq \text{null}) \land (\text{col}_i \notin w_i \lor w_i < t_i >) \) then
3: \( \text{col}_i \leftarrow \text{null} \)
4: \( \text{vers}_i \leftarrow -1 \)
5: \( w_i \leftarrow \emptyset \)
6: \( \text{timer}_i \leftarrow T_{MAX} \)
7: end if
8: Let \( M \) be the set received messages presents in the reception buffer of \( i \).
   // Sort by type all messages in \( M \)
9: Let \( \{\text{M}_TOK, \text{M}_DIV, \text{M}_DISS, \text{M}_WAVE\} \) a partition of \( M \) such that:
   \( \text{M}_TOK \)\{\( m \in M : m \) is a Token message \}
   \( \text{M}_DIV \)\{\( m \in M : m \) is a Division message \}
   \( \text{M}_DISS \)\{\( m \in M : m \) is a Dissolution message \}
   \( \text{M}_WAVE \)\{\( m \in M : m \) is a Reloading_Wave message \}
   // Sort by color all tokens in \( \text{M}_TOK \)
10: Let \( \text{nbCol} \) be the number of different colors among all token colors in \( \text{M}_TOK \).
   Let \( \{\text{M}_TOK\_SORT = \{\text{TOK}_1, \ldots, \text{TOK}_{\text{nbCol}}\}\} \) be a partition of the set \( \text{M}_TOK \) such that:
   \( \forall c, c' \in \{1, \ldots, \text{nbCol}\}, \forall t \in \text{TOK}_c, \forall t' \in \text{TOK}_{c'}: c = c' \iff \text{col}_c = \text{col}_t \)
   // Management of all messages
11: \( \forall TOK \in \text{M}_TOK\_SORT : \text{Manage}\_\text{token}(TOK) \)
12: \( \forall msg \in \text{M}_DIV \_\text{DIV} : \text{Manage}\_\text{Division}(msg) \)
13: \( \forall msg \in \text{M}_DISS : \text{Manage}\_\text{Dissolution}(msg) \)
14: \( \forall msg \in \text{M}_WAVE : \text{Manage}\_\text{Reloading}\_\text{Wave}(msg) \)
   // Timer expiration: new token creation
15: if \( \text{timer}_i \leq 0 \) then
16: if \( \text{col}_i = \text{null} \) then
17: \( \text{col}_i \leftarrow i \)
18: \( w_i \leftarrow < i > \)
19: end if
20: \( \text{vers}_i \leftarrow \text{vers}_i + 1 \)
21: \( \text{timer}_i \leftarrow T_{MAX} \)
22: Random choice of \( x \in N_i \)
23: Send Token(\( \text{col}_i, \text{vers}_i, w_i, 0 \)) to \( x \)
24: end if

The algorithm has only one guarded rule (algorithm 1): when a node wakes up (every bounded period), the node first check its local state to detect some inconsistencies (lines 2 to 7) and becomes a free node if there is some. Then the node
manages of its received messages. This consist of sorting the
messages by type and for Token type messages, the node also
sorts the messages by color (lines 8 to 10). The node calls
some functions to manage messages accordingly to the type
of the message. Each message is managed separately (lines
12 to 14) but token type messages: these one are managed
in group composed by messages of the same color (line 11).
Finally, the node checks if its timer is 0 and creates a new
token if this is the case (lines 15 to 24).

D. Messages management

First, note that all procedures are executed by node i.
The algorithm uses four types of messages: Token, Reload-
ing_Wave, Dissolution and Division.
The token type messages are the main messages of
the algorithm. They represents a cluster and collect nodes during
there circulation.
The three other types of messages are control messages
circulation in a cluster built by a token.
Reloading_Wave messages are used to avoid useless token
creation as we saw in section III-B.
Dissolution messages are used to transform all nodes in a
cluster into free nodes (algorithm 3).
Division messages are used to divide one big cluster (of
size > M) into two smaller clusters (algorithm 4).
Finally, a control message is a message intended for all
nodes in a cluster and gives them an order (reset the timer,
become free, change of cluster). So when a node receives a
control message, the node first checks if the cluster concerned
by this message really is the cluster it belongs to (algorithm 2,
3 and 4, line 1). If this is the case, the node follows the order
and then broadcasts it algorithm 2, lines 7 to 9, algorithm 3,
lines 9 to 11 and algorithm 4, lines 23 to 25).

1) Reloading_Wave management: A Reloading_Wave
message has three variables:
- $col_{RW}$: color of the cluster concerned by the reloading
  wave
- $vers_{RW}$: version of the cluster concerned by the reloading
  wave
- $w_{RW}$: word of the cluster concerned by the reloading
  wave

Algorithm 2 Procedure Manage_Reloading_Wave
(RW Reloading_Wave message)

Require: $RW = Reloading_Wave(w_{RW}, col_{RW}, vers_{RW})$
1: if ($col_{i} = col_{RW} \land vers_{i} \leq vers_{RW}$) then
2:     $vers_{i} \leftarrow vers_{RW}$
3:     timer$_{i} \leftarrow T_{MAX}$
4:     $w_{RW} \leftarrow Clean_Word(w_{RW})$
5:     $A \leftarrow Init_Tree(w_{RW}[1])$
6:     $A \leftarrow Increase_Tree(A, w_{RW})$
7:     for all $j \in My_Son(A)$ do
8:         Send $Reloading_Wave(w_{RW}, col_{RW}, vers_{RW})$ to $j$
9:     end for
10: end if

2) Dissolution management: A Dissolution message has
three variables:
- $col_{Diss}$: color of the cluster concerned by the dissolution
- $vers_{Diss}$: version of the cluster concerned by the dissolution
- $w_{Diss}$: word of the cluster concerned by the dissolution

Algorithm 3 Procedure Manage_Dissolution
(DISS Dissolution message)

Require: $DISS = Dissolution(w_{Diss}, col_{Diss}, vers_{Diss})$
1: if ($col_{i} = col_{Diss} \land vers_{i} \leq vers_{Diss}$) then
2:     $vers_{i} \leftarrow -1$
3:     $w_{i} \leftarrow \emptyset$
4:     timer$_{i} \leftarrow T_{MAX}$
5:     $w_{Diss} \leftarrow Clean_Word(w_{Diss})$
6:     $A \leftarrow Init_Tree(w_{Diss}[1])$
7:     $A \leftarrow Increase_Tree(A, w_{Diss})$
8:     for all $j \in My_Son(A)$ do
9:         Send $Dissolution(w_{Diss}, col_{Diss}, vers_{Diss})$ to $j$
10:     end for
11: end if

Algorithm 4 Procedure Manage_Division
(DIV Division message)

Require: $DIV = Division(w_{DIV}, w_{1}, w_{2}, col_{DIV}, vers_{DIV}, col_{Maj},$ $vers_{Maj}, pivot)$
1: if ($col_{i} = col_{DIV} \land vers_{i} \leq vers_{DIV}$) then
2:     if pivot = $i$ then
3:         $col_{Maj} \leftarrow i$
4:         $vers_{Maj} \leftarrow vers_{i}$
5:         timer$_{i} \leftarrow M$
6:     else
7:         timer$_{i} \leftarrow T_{MAX} + M$
8:     end if
9:     $col_{i} \leftarrow col_{Maj}$
10:     $vers_{i} \leftarrow vers_{Maj}$
11: if $i \in w_{1}$ then
12:     $w_{1} \leftarrow Clean_Word(w_{1})$
13:     $w_{1} \leftarrow Change_First(w_{1}, i)$
14:     $w_{2} \leftarrow w_{1}$
15:     else
16:     $w_{2} \leftarrow Clean_Word(w_{2})$
17:     $w_{2} \leftarrow Change_First(w_{2}, i)$
18:     $w_{1} \leftarrow w_{2}$
19:     end if
20:     $w_{DIV} \leftarrow Clean_Word(w_{DIV})$
21:     $A \leftarrow Init_Tree(w_{DIV}[1])$
22:     $A \leftarrow Increase_Tree(A, w_{DIV})$
23:     for all $j \in My_Son(A)$ do
24:         Send $Division(w_{DIV}, w_{1}, w_{2}, col_{DIV}, vers_{DIV}, col_{Maj},$ $vers_{Maj}, pivot)$ to $j$
25:     end for
26: end if

3) Division management: A Division message has eight
variables:
- $col_{DIV}$: color of the cluster concerned by the division
- $vers_{DIV}$: version of the cluster concerned by the division
- $w_{DIV}$: word of the cluster concerned by the division
- $w_{1}$: word containing all nodes in the first subset of the
  cluster of color $col_{DIV}$
- $w_{2}$: word containing all nodes in the second subset of the
  cluster of color $col_{DIV}$
- $pivot$: identifier of the node that separates the first subset
  from the second one. This node belongs to the second subset.
• \( \text{col}_{	ext{Maj}} \) : color of the current subset (between \( w_1 \) and \( w_2 \)) of the cluster of color \( \text{col}_{	ext{Div}} \)

• \( \text{vers}_{	ext{Maj}} \) : version of the current subset (between \( w_1 \) and \( w_2 \)) of the cluster of color \( \text{col}_{	ext{Div}} \)

4) Token management: We already saw variables in a token. We are now going to see how a node manages a set of tokens of the same color. Let us assume this color is red. There are then 3 possible cases when the \( i \) receives these token messages:

a) If \( i \) is red (i.e. \( i \) belongs to a red cluster):

(i) If the sender is red too:

• \( i \) adds its own ID at the beginning of the word and checks if all neighbors of \( i \) in the word are correct.

• then \( i \) sends the token to one of its neighbor chooses uniformly at random.

(ii) If the sender is free: ignore the token

(iii) If the sender is blue (different from red): the same as in case (a.i)

b) If \( i \) is blue:

(i) If the sender is red:

• \text{PARTITION CORRECTION, i.e.} \( i \) checks if not belong to the word. If this is the case, it changes the word only keeping the prefix that does not contain \( i \).

• if the cluster of the token is bigger than \( m \) then \( i \) sends back the token to its sender, otherwise \( i \) sends back a dissolution message to the sender

(ii) If the sender is free or blue: ignore the token

c) If \( i \) is free:

(i) If the sender is red:

• \text{PARTITION CORRECTION}

• then \( i \) adds its own ID at the beginning of the word and joins the cluster

• if the cluster of the token is smaller than \( M \) then \( i \) sends the token to one of its neighbor chooses uniformly at random, otherwise \( i \) sends back a division message to the sender

(ii) If the sender is free or blue: ignore the token

5) Other procedures:

• Procedure \( \text{Size}(w \text{ word}) : \text{integer} \) – returns the size of the word \( w \)

• Procedure \( \text{Nb_Identities}(w \text{ word}) : \text{integer} \) – returns the number of different identities contained in the word \( w \)

• Procedure \( \text{Delete_Element}(w \text{ word}, z \text{ integer}) : \text{void} \) – deletes the element at the position \( z \) in the word \( w \)

• Procedure \( \text{Reverse}(w \text{ word}) : \text{word} \) – reverses the word \( w \) (i.e. the word \( < w[1], \ldots, w[\text{Size}(w)] \) becomes \( < w[\text{Size}(w)], \ldots, w[1] > \))

• Procedure \( \text{Add_Begin}(j \text{ identifier}, w \text{ word}) : \text{word} \) – returns the word \( < j, w[1], \ldots, w[\text{Size}(w)] > \)

• Procedure \( \text{Left}(z \text{ integer}, w \text{ word}) : \text{word} \) – returns the word \( < w[1], \ldots, w[z] > \)

• Procedure \( \text{Right}(z \text{ integer}, w \text{ word}) : \text{word} \) – returns the word \( < w[z+1], \ldots, w[\text{Size}(w)] > \)

• Procedure \( \text{Init_Tree}(j \text{ identifier}) : \text{tree} \) – returns a tree only composed by a root \( j \)
• Procedure \textit{Add\_Son(}A \textit{tree, }j \textit{identifier, }k \textit{identifier)} : \textit{tree} — adds \textit{j} to the set of \textit{k} sons in the tree \textit{A}.

• Procedure \textit{My\_Son(}A \textit{tree)} : \textit{set of identifiers} — returns the set of son in \textit{A} of the node executing the procedure.

• Procedure \textit{Clean\_Word(}w \textit{word)} : \textit{word} — deletes all inconsistencies about \textit{i}'s neighborhood and deletes all successive occurrences of \textit{i} in the word.

• Procedure \textit{Tree\_To\_Word(}A \textit{tree)} : \textit{word} — transforms a tree \textit{A} into a word \textit{w} such that the first identifier of \textit{w} is the root of \textit{A}.

• Procedure \textit{Change\_First(}w \textit{word, }j \textit{identifier)} : \textit{word} — the returned word is such that (i) \textit{j} is the first identifier and (ii) the tree transformation of \textit{w} and of the returned word are the same except there are not enrooted on the same node.

• Procedure \textit{Increase\_Tree(}A \textit{tree, w word)} : \textit{tree} — requires \textit{A} root to be the same than the first identifier of \textit{w}. Returns the tree \textit{A} where all new links contained in \textit{w} have been added.

• Procedure \textit{Words\_Fusion(}W \textit{set of words, }c \textit{color)} : \textit{word} — requires \textit{c} to belong to each word in \textit{W}. The resulted word is such that (i) the tree we can build from it is a spanning tree of all nodes contained in \textit{W} and (ii) the first identifier of it is a neighbor of \textit{i}.

\section*{References}


