International Journal of Networking and Computing – `www.ijnc.org`, ISSN 2185-2847
Volume 8, Number 1, pages 53–72, January 2018

A Self-Stabilizing Algorithm for Constructing a Maximal (1,1)-Directed Acyclic Mixed Graph[1]

Yonghwan Kim, Haruka Ohno, Yoshiaki Katayama
Graduate School of Engineering, Nagoya Institute of Technology
Aichi, 466–8555, Japan
Email: kim@nitech.ac.jp, ohno@moss.elcom.nitech.ac.jp, katayama@nitech.ac.jp

and

Toshimitsu Masuzawa
Graduate School of Information and Technology, Osaka University
Osaka, 565–0871, Japan
Email: masuzawa@ist.osaka-u.ac.jp

Received: July 25, 2017
Revised: September 25, 2017
Accepted: November 29, 2017
Communicated by Akihiro Fujiwara

Abstract

We introduce a new network structure named a maximal (σ, τ)-directed acyclic mixed graph (DAMG). A maximal (σ, τ)-DAMG allows both arcs (directed edges) and (undirected) edges which is constructed, for any given connected undirected graph with a set of σ nodes specified as source nodes and a set of τ nodes specified as sink nodes, by assigning directions to as many (undirected) edges as possible (i.e., by changing edges into arcs) so that the following conditions are satisfied: (i) each node specified as a source node has at least one outgoing arc but no incoming arc, (ii) each node specified as a sink node has at least one incoming arc but no outgoing arc, (iii) each other node has no arc or has both outgoing and incoming arcs, and (iv) no directed cycle (consisting only of arcs) exists. This maximality implies that changing any more edges to arcs violates these conditions, for example, a source node has an incoming arc, a node which is specified as neither a source node nor a sink node has only outgoing or incoming arcs other than edges, or a directed cycle is created in the network.

In this paper, we propose a self-stabilizing algorithm which constructs a (1,1)-maximal DAMG in any connected graph with a specified source node and a specified sink node by assigning directions to as many edges as possible.

Keywords: Directed Acyclic Mixed Graph, Self-Stabilizing Algorithm, Maximal DAMG Construction

[1]The preliminary version of this paper appeared in the proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW), which entitled *A Self-Stabilizing Algorithm for Constructing (1,1)-Maximal Directed Acyclic Graph.*

1 Introduction

1.1 Background

A directed acyclic graph (DAG) is a directed graph that has no cycle. Any DAG has at least one node called a source that has no incoming arc and at least one node called a sink that has no outgoing arc. Each source node has a directed path to a sink node (but not necessarily to every sink node). Therefore, communication routes can be ensured from source nodes (can be the senders of messages) to sink nodes (can be the receivers of the messages) by constructing a DAG in any connected network.

However, if some nodes are predetermined as either source nodes or sink nodes in an arbitrary (undirected) graph, a DAG may not be constructed consistently. This is because it is impossible to make the predetermined source nodes (resp. sink nodes) have only outgoing (incoming) edges and make the other nodes have both incoming and outgoing arcs. For example, given a tree with predetermined source and sink nodes, a leaf node becomes a source or sink node even when it is not a predetermined source or sink. Therefore, a consistent DAG construction can not be guaranteed from any graph with predetermined source nodes and sink nodes unless we allow some edges to remain undirected.

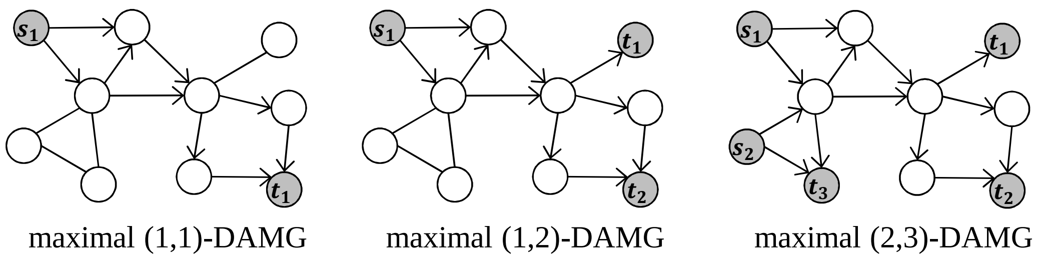


Figure 1: Examples of maximal (σ, τ)-DAMGs

Given a connected network with predetermined σ source nodes and τ sink nodes, we formalize a consistent DAG (possibly with both arcs and edges) as a new network structure named a (σ, τ)-Directed Acyclic Mixed Graph (DAMG). A (σ, τ)-DAMG is a mixed graph which allows both arcs (or directed edges) and (undirected) edges such that there exist exactly σ source nodes and τ sink nodes but there exists no directed cycle (consisting of only arcs). Each source (resp. sink) node has at least one outgoing (resp. incoming) arc but no incoming (resp. outgoing) arc. Moreover any other node is neither a source nor a sink node; it has no arc or both outgoing and incoming arcs.

This paper considers maximal (σ, τ)-DAMG construction: A maximal (σ, τ)-DAMG is constructed by assigning directions to as many edges as possible so that the following conditions should be satisfied: (i) each node specified as a source node has at least one outgoing arc but no incoming arc, (ii) each node specified as a sink node has at least one incoming arc but no outgoing arc, (iii) each other node has no arc or has both outgoing and incoming arcs, and (iv) no directed cycle (consisting only of arcs) exists. Thus if any of the remaining edges are assigned directions, the resultant graph becomes no longer a consistent DAMG (violates any of above conditions). A maximal (σ, τ)-DAMG can be also presented as a maximal (S, T)-DAMG using a set representation when a undirected graph G and two disjoint sets $S(\subset G)$ and $T(\subset G)$ are given.

Figure 1 shows the examples of some maximal (σ, τ)-DAMGs, which are a maximal (1,1)-DAMG, a maximal (1,2)-DAMG, and a maximal $(2, 3)$-DAMG. These can be also represented as a maximal $(\{s_1\}, \{t_1\})$-DAMG, a maximal $(\{s_1\}, \{t_1, t_2\})$-DAMG, and a maximal $(\{s_1, s_2\}, \{t_1, t_2, t_3\})$-DAMG respectively.

In this paper, we propose a self-stabilizing algorithm for constructing a maximal (1,1)-DAMG in any connected undirected network when only a single node is specified as a source node and another single node is specified as a sink node.

A self-stabilizing algorithm is a distributed algorithm which manages to eventually present an appropriate behavior starting from any initial configuration [3, 5]. From this property, a self-stabilizing algorithm guarantees that the system can be recovered to its intended configuration within a finite time even if any number of transient failures occur and bring the system to any inconsistent configuration. Therefore, a self-stabilizing algorithm can be operated from any initial state and the system is always recovered to a consistent configuration, thus it can be applied to the system which freely changes its configuration.

1.2 Related Works

A transport net of any biconnected network is a DAG (only with arcs) which has two distinct predetermined nodes, a source node and a sink node. In a transport net, every node other than the source node and the sink node has both incoming arcs and outgoing arcs (Fig. 2(a)). The maximal (1,1)-DAMG considered in this paper is a generalization of the transport net in the sense that any maximal (1,1)-DAMG is a transport net when the given network is biconnected.

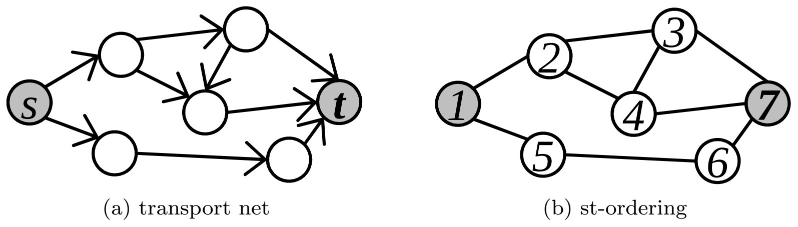


Figure 2: A transport net and st-ordering

An st-ordering (also known as st-numbering) is one of the related works of a transport net. An st-ordering is assignment of an st-order (number) to each node. The predetermined source node is assigned 1, and the predetermined sink node is assigned n (where n is the total number of nodes), the other nodes are assigned distinct integers ranging from 2 to $(n-1)$ so that the st-numbering represents a topological order of a transport net (Fig. 2(b)). This implies that a transport net can be constructed if an st-order can be assigned.

J. Ebert had proposed a sequential algorithm to find an st-ordering [6]. This algorithm starts with constructing a depth-first-search (DFS) tree rooted at the source node, and maintains the list derived from the tree path from the source node to the sink node and back edges of the tree. An st-ordering can be found using this list. Aranha et. al. had introduced a distributed algorithm to find an st-ordering [1] which is based on [6].

Chaudhuri et. al. expanded the st-ordering algorithm[6] to a self-stabilizing algorithm [2]. It constructs a DFS tree rooted at the source node, and makes the path list of the tree path from the source node to the sink node. After that, each node updates the path information using the information of its adjacent nodes. An st-ordering can be realized using the relations between nodes. Karaata et. al. had proposed another self-stabilizing st-order algorithm [7] using two breadth-first search (BFS) trees rooted at the source node and the sink node respectively. All nodes in the network are divided into two sets: one consists of the nodes closer to the sink node than the source node, and the other consists of the remaining nodes. A transport net can be constructed by directing each edge from the source node to the sink node and updating the directions of edges of end nodes except the sink node.

1.3 Organization of the Paper

This paper is organized as follows: Section 2 introduces our system model and the problem we consider. An algorithm for constructing a maximal (1,1)-DAMG and its correctness are presented in Section 3. A summary and future works are given in Section 4.

2 System Model and Problem Definition

In this paper, we consider a maximal (σ, τ)-DAMG construction problem and an algorithm which constructs a maximal (σ, τ)-DAMG. When an arbitrary connected undirected graph is given and σ nodes are specified as source nodes and τ nodes are specified as sink nodes, we consider how to construct a valid maximal (σ, τ)-DAMG by assigning directions to edges.

To discuss our algorithm which constructs a maximal (σ, τ)-DAMG, we introduce our system model and a formal definition of a problem for constructing a maximal (σ, τ)-DAMG in this section.

2.1 Undirected Graph

An undirected graph G consists of a non-empty set of nodes (also called vertex) V and a set of edges E, and we notate $G = (V, E)$. If there is an edge between nodes u and v, we call u and v are adjacent and notate $(u, v) \in E$.

A sequence $\langle v_0, v_1, \cdots, v_m \rangle$ consisting of distinct nodes of G when $(v_i, v_{i+1}) \in E$ for every i (where $0 \leq i < m$) is called a $v_0 - v_m$ path. If both a $v_0 - v_m$ $(m \geq 2)$ path and edge (v_m, v_0) exist ($(v_m, v_0) \in E$), we call a sequence $\langle v_0, v_1, \cdots, v_m, v_0 \rangle$ a cycle. G is connected if there is a $v_i - v_j$ path for any distinct nodes v_i and v_j. Otherwise, G is disconnected. If there are two or more internally disjoint $v_i - v_j$ paths for any distinct nodes v_i and v_j in G, G is called biconnected.

Definition 1. *(Articulation Point) A node $v \in V$ is an articulation point of a connected graph $G = (V, E)$ when G becomes disconnected if v is removed from G.*

A connected graph which has no cycle is called a tree. When G is a tree, a node having only one adjacent node is called a leaf node.

2.2 Depth-First Search (DFS) Tree

Depth-first search (DFS) is a method for searching a graph which traverses as far as possible along edges. DFS starts from an initial vertex v (v is initially explored) and explores an unexplored adjacent node. This process is repeated as long as the currently visited node has an unexplored adjacent node. If the currently visited node has no unexplored adjacent node, it backtracks along the edge along which it first visited the node until it goes back to a node that has an unexplored adjacent node. A tree which is constructed by DFS of a graph G is called a DFS tree.

2.3 Directed Graph and DAG

A directed graph $\overrightarrow{G}$ can be represented as $\overrightarrow{G} = (V, \overrightarrow{E})$. V is the non-empty set of vertices, and $\overrightarrow{E}$ is the set of arcs. If there is an arc from a node v_i to a node v_j, $\overrightarrow{(v_i, v_j)} \in \overrightarrow{E}$. When $\overrightarrow{(v_i, v_j)} \in \overrightarrow{E}$, $\overrightarrow{(v_i, v_j)}$ is called an outgoing arc of v_i and an incoming arc of v_j. A sequence $\langle v_0, v_1, \cdots, v_m \rangle$ consisting of distinct nodes of $\overrightarrow{G}$ when $\overrightarrow{(v_i, v_{i+1})} \in \overrightarrow{E}$ for every i (where $0 \leq i < m$) is called a directed $v_0 \to v_m$ path. If both a directed $v_0 \to v_m$ path of length 2 or more and $\overrightarrow{(v_m, v_0)}$ exist, we call $\langle v_0, v_1, \cdots, v_m, v_0 \rangle$ a directed cycle.

A directed graph which has no directed cycle is called a directed acyclic graph (DAG). A node which has no incoming arc is called a *source* node and a node which has no outgoing arc is called a *sink* node. Any DAG has at least one source node and at least one sink node.

2.4 Transport Net Problem and st-ordering Problem

Let $G = (V, E)$ be any biconnected undirected graph where two distinct nodes s and t are given. A transport net problem is to construct a DAG from G by assigning all edges directions so that only s becomes a source node and only t becomes a sink node [2, 6].

An st-ordering problem for G is to assign st-order (numbers) to all nodes in G as follows: s is assigned 1, t is assigned n (where n is the total number of the nodes) and each other nodes is assigned a distinct integer ranging from 2 to $(n - 1)$ so that it has an adjacent node with a smaller st-number and an adjacent node with a larger st-number. In other words, the st-numbers assigned to nodes are consistent with a topological order of a transport net of G. This implies that if the st-ordering problem can be resolved, a transport net problem can be also resolved.

2.5 Network and Processes

In this paper, we assume a connected network N of arbitrary topology consisting of n processes. Each process in N operates asynchronously, and two distinct processes are predetermined as a source node and a sink node.

Let $P = \{p_1, p_2, \cdots, p_n\}$ be the set of n processes, and L be the set of the communication links. In this case, we denote $N = (P, L)$.

We assume p_1 is a source node and p_n is a sink node, and these two nodes have unique identifiers. We denote a source node and a sink node, s and t respectively. Each process in $\{p_2, p_3, \cdots, p_{n-1}\}$ is anonymous (we use subscript notation from 2 to $n-1$ only for explanation). If $(p_i, p_j) \in L$ (where, $1 \leq i < j \leq n$), there is a full duplex link between p_i and p_j. A process p_i maintains a port number of each link incident to p_i which is totally ordered at p_i and can identify each link by the port number.

We assume a register communication model where a process p_i communicates with a process p_j using shared registers R_{ij} and R_{ji}. A register R_{ij} is written by p_i and is read by p_j and vice versa. Writing and reading operations are executed by calling the functions *write* and *read* respectively. We define two functions *read* and *write* which can be called by p_i as follows.

Definition 2. *(Functions read and write)*

- *read*(R_{ji}) *: A function returns a value which is read from register* R_{ji}*.*
- *write*(R_{ij}, x) *: A function writes* x *to register* R_{ij}*.*

Note that a network $N = (P, L)$ can be handled as a graph. Thus, a process and a node are used interchangeably and a link and an edge are also.

2.6 Schedule

Let q_i be the state of a process p_i. A configuration (or a global state) c of the network N is represented as $c = (q_1, q_2, \cdots, q_n)$ where q_i includes all data of the registers written by p_i in addition to the internal state of p_i. C is the set of all possible configurations of N, thus, when Q_i is the set of all possible states of p_i, C can be described as $C = Q_1 \times Q_2 \times \cdots \times Q_n$.

Let S be a subset of P, and $\mathscr{A}$ be an algorithm. Let c be an arbitrary configuration ($c \in C$). When each process in S operates an atomic action of $\mathscr{A}$, the configuration of the network c is changed into c', then we notate this as $c \mapsto (S, \mathscr{A})c'$.

Definition 3. *(Schedule and Execution) We call a sequence* $T = S_0, S_1, S_2, \cdots$ *of non-empty process sets a schedule. In infinite sequence* $E = (c_0, c_1, c_2, \cdots)$ *of network configurations, if* $c_i \mapsto (S_i, \mathscr{A})c_{i+1} (i \geq 0)$ *is satisfied, we call* E *the execution of algorithm* $\mathscr{A}$ *along the schedule* T *from the initial configuration* c_0*.*

Definition 4. *(Fair Schedule) If every process* $p_i \in P$ *appears infinitely often in a schedule* T*, we call the schedule* T *a fair schedule.*

We consider only fair schedules in this paper unless specifically mentioned. Each process operates one atomic action when it is selected in a schedule. There are some scheduler models depending on the number of concurrently operating processes and granularity of an atomic action. In this paper, we consider the distributed daemon[4, 10] which assumes the followings:

- The number of the process : $|S_i| \geq 1$ for any $i (\geq 0)$.
- An atomic action : Read data from all adjacent registers, update its internal state, and write data to all adjacent registers.

2.7 Self-Stabilizing Algorithm

An algorithm $\mathscr{A}$ is a self-stabilizing algorithm for $C_{\mathscr{L}\mathscr{E}} \subset C$ and is denoted $SS(\mathscr{A}, C_{\mathscr{L}\mathscr{E}})$ if the following two conditions hold. We call any configuration in $C_{\mathscr{L}\mathscr{E}}$ a consistent configuration for algorithm $\mathscr{A}$. If $\mathscr{A}$ is obvious, we just call it a *consistent configuration.*

1. **Convergence**: For an arbitrary configuration of the network $c \in C$ and an arbitrary schedule T, there is $c' \in C_{\mathscr{L}\mathscr{E}}$ which appears in the execution of $\mathscr{A}$ with a schedule T.
2. **Closure**: For an arbitrary configuration $c \in C_{\mathscr{L}\mathscr{E}}$, if $c \mapsto c'$, then c' is included in $C_{\mathscr{L}\mathscr{E}}$.

The above conditions imply that starting from any configuration $c \in C$, algorithm $\mathscr{A}$ reaches a consistent configuration $c' \in C_{\mathscr{L}\mathscr{E}}$ within finite time, and remains in consistent configurations after reaching c'.

2.8 Fair composition

A fair composition is a composition of self-stabilizing algorithms. When self-stabilizing algorithms $\mathscr{A}_1, \mathscr{A}_2, \cdots, \mathscr{A}_k$ are converged, the output of $\mathscr{A}_k$ can be the input of $\mathscr{A}_{k+1}$[5]. $\mathscr{A}_{k+1}$ can be executed whether algorithms $\mathscr{A}_1, \mathscr{A}_2, \cdots, \mathscr{A}_k$ are converged or not. After the convergence of algorithms $\mathscr{A}_1, \mathscr{A}_2, \cdots, \mathscr{A}_k$, algorithm $\mathscr{A}_{k+1}$ begins to converge to a consistent configuration. When a fair composition combines two algorithms $\mathscr{A}_1$ and $\mathscr{A}_2$, it satisfies the following conditions:

- A variable which can be written by $\mathscr{A}_1$ can be read by $\mathscr{A}_2$ but cannot be written by $\mathscr{A}_2$.
- A variable which can be written by $\mathscr{A}_2$ is never read or written by $\mathscr{A}_1$.

When a process executes an atomic action, it executes an atomic action of each of $\mathscr{A}_1$ and $\mathscr{A}_2$ in this order. In a fair composition, if $\mathscr{A}_1$ converges to a consistent configuration, $\mathscr{A}_2$ will converge to a consistent configuration within finite time. Obviously, a fair composition can be applied to three or more algorithms.

2.9 maximal (σ, τ)-DAMG construction Problem

We define a maximal (σ, τ)-DAMG construction problem as follows.

Definition 5. *Let $G = (V, E)$ be an arbitrary undirected graph where σ nodes in the set $S(\subset V)$ are specified as source nodes and τ nodes in the set $T(\subset V)$ are specified as sink nodes. A maximal (σ, τ)-DAMG is a mixed graph (consisting of both arcs and edges) obtained from G by assigning directions to some edges of G (i.e., replacing some undirected edges of G with arcs) so that the following conditions are satisfied.*

1. *Each node $s \in S$ is a source node (or it has no incoming arc) having at least one outgoing arc and each node $t \in T$ is a sink node (or it has no outgoing arc) having at least one incoming arc.*
2. *Each node $v \in (V \backslash (S \cup T))$ is neither a source node nor sink node (i.e., it has no arc or it has both an outgoing arc and an incoming arc).*
3. *No directed cycle exists.*
4. *If any one or more edges are further changed into arcs, at least one of the above three conditions is violated.*

We call a *maximal* DAMG because of the condition 4 and this implies that a maximal (σ, τ)-DAMG has a maximal number of arcs. However there are some cases that no DAMG constructed from G can satisfy all the above conditions. For example, when three nodes in S form a line subgraph, at least one of the edges adjacent to the center node (say v) has to be changed to an outgoing arc

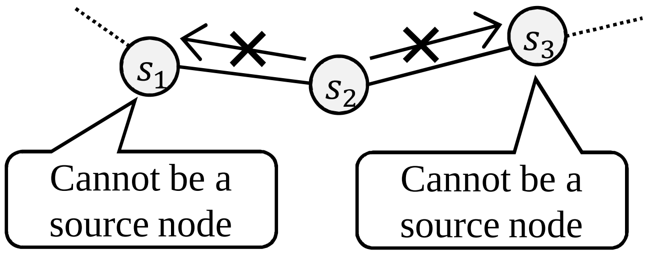


Figure 3: An example for all nodes in S cannot be source nodes

of v. But it makes the other end node (say u) have an incoming arc and thus u cannot be a source node (Fig. 3). The same holds for T.

A problem which constructs a maximal (σ, τ)-DAMG (if possible) from any given undirected graph G with specified source nodes S and sink nodes T is called a maximal (σ, τ)-DAMG construction problem. In this paper, we propose a self-stabilizing algorithm for the maximal (1,1)-DAMG construction problem which handles one source node s and one sink node t only. Notice that a maximal (1,1)-DAMG can be constructed from any connected graph.

3 Algorithm for constructing a maximal (1,1)-DAMG

In this section, we present our self-stabilizing algorithm for constructing a maximal (1,1)-DAMG in an arbitrary anonymous connected network $N = (P, L)$ which has two specific nodes s and t.

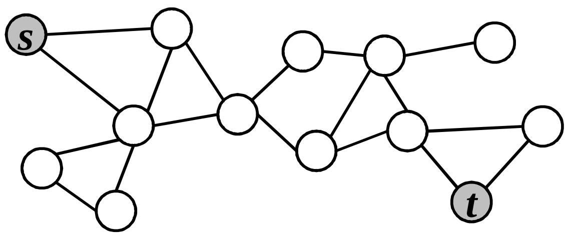
3.1 Overview of our proposed algorithm

In our algorithm, we divide the initial network N into some biconnected subgraphs which are connected by articulation points. We call these biconnected subgraphs biconnected blocks. Moreover, our algorithm constructs the DFS tree rooted at s on the initial network N. We denote a path from s to t $s - t$ and a path from s to t in a DFS tree rooted at s $s \overset{*}{-} t$. Note that $s \overset{*}{-} t$ is the special case of a $s - t$ path.

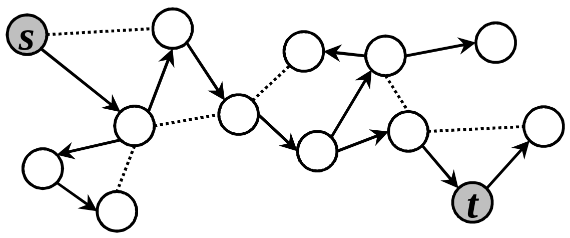
Figure 4 shows an example of a maximal (1,1)-DAMG constructed on N. Fig. 4(a) illustrates an initial network N and a DFS tree constructed on N is described in Fig. 4(b). Note that there is the only one path from s to t ($s \overset{*}{-} t$) in a DFS tree. Fig. 4(c) shows biconnected blocks which are connected at articulation points.

In every biconnected block including an edge of the $s \overset{*}{-} t$ path, every edge is assigned a direction so that a transport net is constructed where the source is the node nearest to s (s or an articulation point) and the sink is the node nearest to t (t or an articulation point). In every other biconnected block (including no edge of the $s \overset{*}{-} t$ path), no edge is assigned a direction. If some edges in the biconnected blocks are assigned directions, some nodes other than s or t will be source or sink nodes or some directed cycles will be created, thus, such biconnected blocks cannot include any arc. From these properties, we design a procedure that constructs a maximal (1,1)-DAMG from an arbitrary network N as follows.

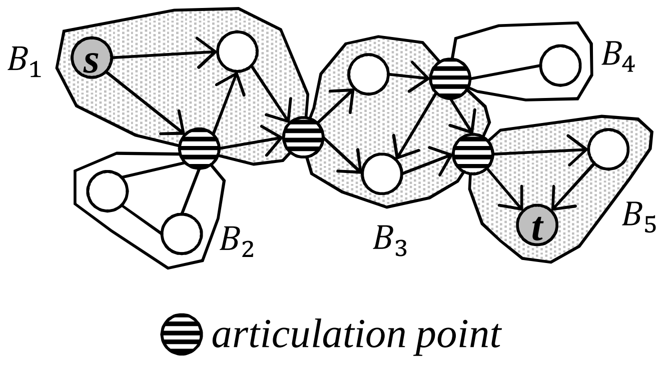
1. Construct a DFS tree rooted at s.
2. Find articulation points, and divide N into biconnected blocks.
3. Find the $s \overset{*}{-} t$ path.
4. Classify biconnect blocks into 2 groups: one consists of all the biconnected blocks including edges of $s \overset{*}{-} t$ and the other consists of the remaining biconnected blocks.
5. Find st-ordering of every biconnected block including an edge of the $s \overset{*}{-} t$ path.



(a) Initial Network with s and t



(b) DFS tree rooted at s



(c) maximal (1,1)-DAMG

Figure 4: An example of a maximal (1,1)-DAMG

6. Construct a transport net based on the st-order in each of the biconnected block.

To implement the above procedure, we combine some self-stabilizing algorithms using the fair composition[5]. More precisely, we use algorithm $DFST$ [9] for step 1, algorithm $FART$ [8] for step 2, and algorithm $stORDER$ [2] for step 5. Therefore, we briefly introduce self-stabilizing algorithms for steps 1, 2 and 5 due to the previous works, we mainly present self-stabilizing algorithms for steps 3, 4 and 6. Figure 5 illustrates an overview of our algorithm. We introduce the algorithms to implement the above procedure in sequence.

3.2 Algorithm $DFST$ for constructing a DFS tree (step 1)

In this subsection, we briefly introduce a previous work, algorithm $DFST$, which constructs a DFS tree rooted at s on the given network $N = (P, L)$ (or graph $G = (V, E)$).

Algorithm $DFST$ uses a traversal order of each node and the number of the decendants of each process. Therefore, algorithm $DFST$ requires only $O(\log n)$ as a message size. A traversal order a natural number provisionally assigned to each process, and it is updated so that it matches the actual number of the traversal order in the depth first search. Each child process sends the number of its descendants to its parent process, and the parent process returns the traversal order of each

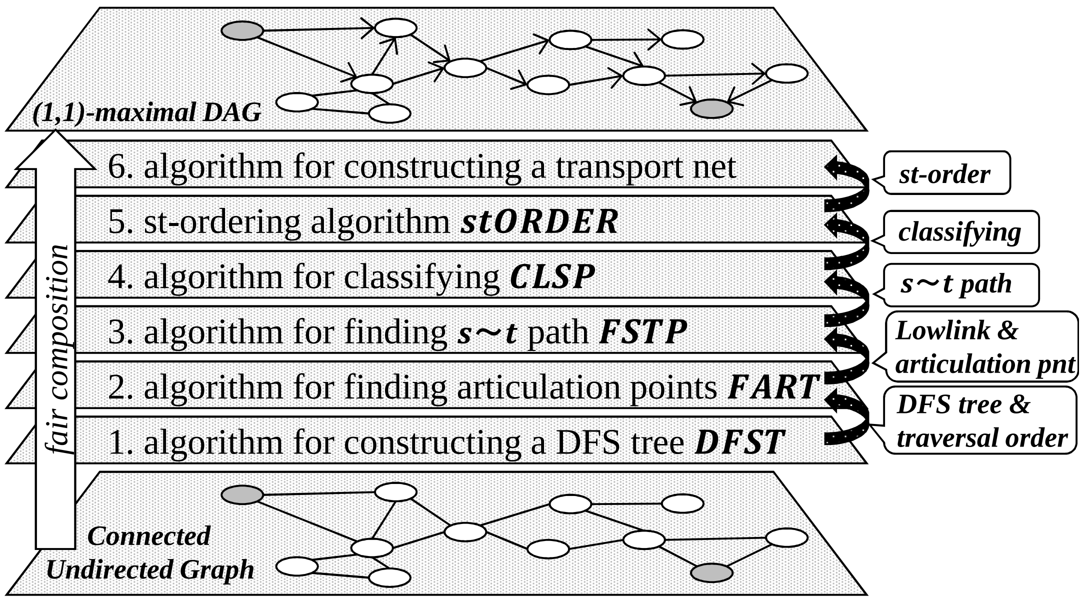

(1,1)-maximal DAG
6. algorithm for constructing a transport net
5. st-ordering algorithm stORDER
4. algorithm for classifying CLSP
3. algorithm for finding s∼t path FSTP
2. algorithm for finding articulation points FART
1. algorithm for constructing a DFS tree DFST
fair composition
st-order
classifying
s∼t path
Lowlink & articulation pnt
DFS tree & traversal order
Connected Undirected Graph

Figure 5: An overview our algorithm

child node which is calculated using the number of each child node's decendants.

The behavior of algorithm $DFST$ is divided depending on whether the process is a source node or not. Trivially, a source node does not have a parent node, thus its traversal order becomes 0.

We show an outline of algorithm $DFST$ as follows:

1. A source node s determines one child node among its neighbor processes, assigns 1 and sends it to the child node.

2. Each process finds its parent process among its neighbor processes, and receives its traversal order.

3. At step 2, if there are two or more parent (candidate) processes, it selects a parent process so that its traversal order becomes larger.

4. If there is no parent (candidate) process among its neighbor processes, it makes its traversal order into $\perp$, and waits until the neighboring process makes it a child node. A process which has $\perp$ as its traversal order becomes a childe node of a process whose traversal order is not $\perp$.

5. Repeat these steps until reaching the leaf node.

6. From the leaf process, return the number of its descendants to its parent node until it reach the root (a source node).

7. A process that receives the number of the decendants from its child node searches for a new child node among its neighbor nodes and sends a new traversal order according to the number of the decendants of its child node.

8. Return to step 2.

For more details, refer [9].

3.3 Algorithm $FART$ for finding articulation points (step 2)

In this subsection, we shortly introduce algorithm $FART$ which finds all articulation points when a DFS tree is constructed on the given network $N = (P, L)$ (or graph $G = (V, E)$).

Algorithm $FART$ uses traversal orders of the DFS tree and a variable *LowLink*. *LowLink* of each node is either its parent node's traversal order or the traversal order of the ancestral process that is reachable without going through its parent process. In this algorithm, each process judges whether it is an articulation point or not using its own traversal order and its *LowLink*. This implies that each process can know whether it is an articulation point or not if it knows *LowLink*. Each process can know its own traversal order of the DFS tree due to a fair composition.

We introduce how to calculate *LowLink* of each node as follows:

1. A source node s set its *LowLink* to 0.

2. The other process finds the minimum traversal order *minTO* among its neighbor processes.

3. If there is a neighbor process p_n which has a smaller traversal order than it, compare its neighbor process' $LowLink_n$ with *minTO* (step 2), and if its neighbor process' $LowLink_n$ is smaller than *minTO*, set $LowLink_n$ to its own *LowLink*, otherwise, set *minTO* to its own *LowLink* (If there are two or more neighbor processes which has a smaller traversal order than itself, finds the minimum *LowLink* among them.)

4. If there is no neighbor process which has a smaller traversal order than it, set *minTO* to its own *LowLink*.

5. Return to step 2.

The value of *LowLink* is determined on each node by executing the above steps repeatedly. Each node can determine whether itself is an articulation point or not using this calculated *LowLink* as following.

- A source node s is an articulation point if it has two or more child nodes.

- The other process p_i is an articulation point if there is a child process whose *LowLink* is the same as p_i's traversal order.

For more details, refer [8].

3.4 Algorithm $FSTP$ for finding a path from s to t (step 3)

In this subsection, we introduce algorithm $FSTP$ for finding the $s \overset{*}{-} t$ path on a DFS tree, when a DFS tree rooted at s is constructed on the given network $N = (P, L)$ (or graph $G = (V, E)$).

Algorithm $FSTP$ is executed on a DFS tree rooted at s, it ensures that each process which appears in the $s \overset{*}{-} t$ path sets its local variable *sink_exists* to *true*, and the other processes set it to *false*. Algorihtm $FSTP$ determines whether each process appears in the $s \overset{*}{-} t$ path or not using the DFS tree; it can refer its parent process and a set of child processes which are obtained in step 1 (algorithm $DFST$ [9]) due to the fair composition.

3.4.1 Overview of algorithm $FSTP$

We show an outline of algorithm $FSTP$ in the following.

1. Process t : Set *sink_exists* to *true*.

2. The other processes : Set *sink_exists* to *true* if it has a child node of which *sink_exists* is *true*, otherwise, set to *false*.

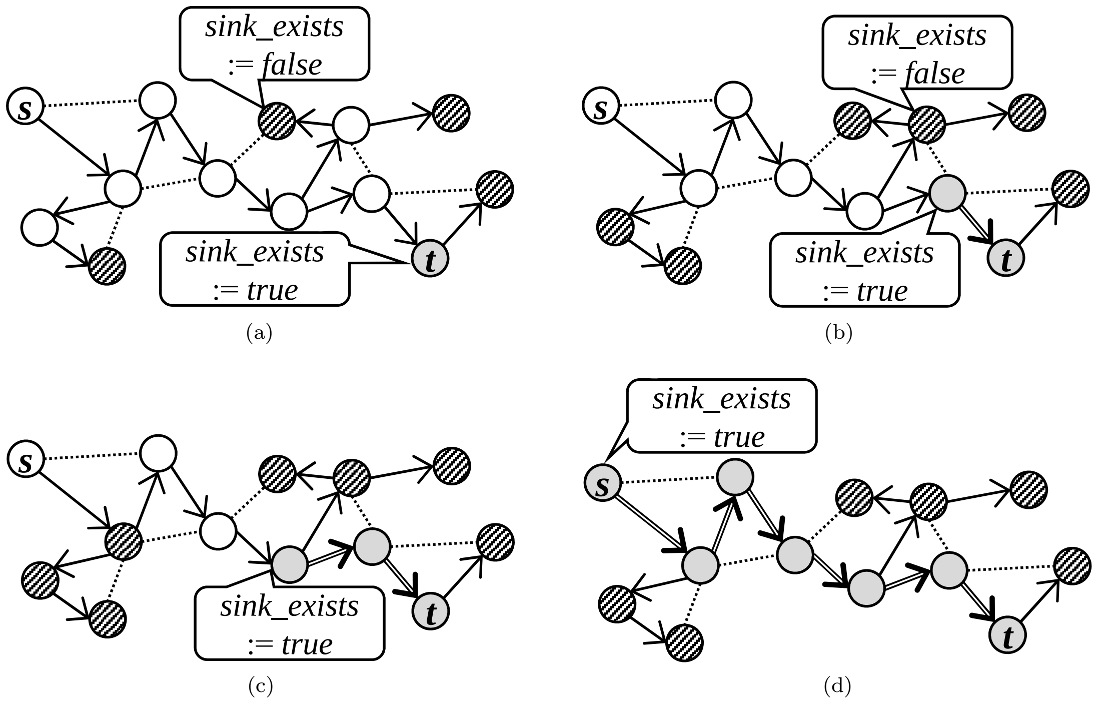


Figure 6: An execution example of algorithm $FSTP$

Figure 6 shows an execution example of algorithm $FSTP$. When algorithm $FSTP$ is executed, t sets its local variable *sink_exists* to *true*. Each process except t set its local variable *sink_exists* to *true* when it has a child of which *sink_exists* is *true*. Therefore, processes appearing in the $s \overset{*}{-} t$ path eventually set their local variable *sink_exists* to *true* in sequence from t to s. The processes that do not appear in the $s \overset{*}{-} t$ path update their local variable *sink_exists* to *false* in the order from leaf processes to their ancestors.

3.4.2 Variables of each process

In this subsection, we introduce local variables which are maintained by each process.

- A variable which is written to a register.
  - $sink\_exists_i \in \{true, false\}$: A boolean variable whether p_i appears in the $s \overset{*}{-} t$ path.
- Constant variables of p_i.
  - N_i: A set of p_i's neighbor processes.
  - $children_i$: A set of the child processes of p_i on a DFS tree. Algorithm $DFST$ [9] sets this variable adequately and algorithm $FSTP$ can read this variable due to the fair composition.

Algorithm 1 shows an implementation of $FSTP$ using the variables introduced above.

3.4.3 Proof of correctness

In this subsection, we prove the correctness of $FSTP$. Firstly we define a consistent configuration as follows.

Algorithm 1 Algorithm $FSTP$ for finding the $s \overset{*}{-} t$ path.

```
 1: struct $x$ { $sink\_exists \in \{true, false\}$ }
 2: if $p_i$ is $t$ then                                  ▷ when $p_i$ is a sink node
 3:     while forever do
 4:         $sink\_exists_i \Leftarrow true$
 5:         for every $p_k \in N_i$ do $write(R_{tk}, sink\_exists_i)$
 6:     end while
 7: else                                                  ▷ when $p_i$ is not a sink node
 8:     while forever do
 9:         for every $k \in N_i$ do $x[k] = read(R_{ki})$
10:         if $\exists p_j (\in children_i), x[j].sink\_exists_j$ then
11:             $sink\_exists_i \Leftarrow true$
12:         else
13:             $sink\_exists_i \Leftarrow false$
14:         end if
15:         for every $p_k \in N_i$ do $write(R_{ik}, sink\_exists_i)$
16:     end while
17: end if
```

Definition 6. *(Consistent configuration of algorithm $FSTP$) Configuration c of N is consistent for $FSTP$ if it fulfills the following two conditions:*

1. *For each process p_i, p_i's local variable sink_exists and each register $R_{ij}(\forall p_j \in N_i)$'s sink_exists have the same value.*

2. *Only the processes in the $s \overset{*}{-} t$ path have their local variable sink_exists set to true.*

Now we prove the correctness of algorithm $FSTP$.

Lemma 1. *Assume that a DFS tree rooted at s is constructed and every process has executed an atomic action of the algorithm at least once. Even if algorithm $FSTP$ starts execution from an arbitrary configuration, each process in the $s \overset{*}{-} t$ path eventually sets its local variable sink_exists to true, and the other processes set sink_exists to false. And these variables will remain unchanged after convergence.*

Proof. Due to the assumption, each process executes an atomic action of the algorithm at least once, therefore, its local variable *sink_exists* and each register's *sink_exists* written by the process have the same value.
Each process in the $s \overset{*}{-} t$ path: t makes its local variable *sink_exists true* by line 4 of Algorithm 1 and converges. After that, t's parent process also makes its *sink_exists true* by lines 10 and 11 of Algorithm 1 and converges. As the same manner, all processes on the $s \overset{*}{-} t$ path update their local variable *sink_exists* to *true* in sequence toward s from t.
The other processes (not in the $s \overset{*}{-} t$ path): Each process which is not in the $s \overset{*}{-} t$ path does not have a child process which eventually keeps its local variable *sink_exists true.* If such a child does not exist, it will make its *sink_exists false* by line 13 of Algorithm 1. Therefore, all processes which are not in the $s \overset{*}{-} t$ path make their *sink_exists false* and converge. □

The following theorem holds from the above Lemma.

Theorem 1. *Algorithm FSTP is a self-stabilizing algorithm which finds the $s \overset{*}{-} t$ path from a DFS tree.*

3.5 Algorithm $CLSP$ for a process classification (step 4)

In this subsection, we propose algorithm $CLSP$ for classifying all processes into two groups, processes in biconnected blocks including edges of the $s \overset{*}{-} t$ path and the others when the path $s \overset{*}{-} t$ and all articulation points are given. Note that we will construct a transport network using st-ordering only on the biconnected blocks which include edges of the $s \overset{*}{-} t$ path. In our algorithm, all processes are classified into three groups as follows (Fig. 7).

- **(st-node)** Processes becoming a source or sink node of the transport net constructed in each of the biconnected blocks including edges of the $s \overset{*}{-} t$ path.
- **(normal)** Processes other than the st-nodes in each of the biconnected blocks including edges of the $s \overset{*}{-} t$ path.
- **(NULL)** The processes other than the st-nodes and the normal nodes.

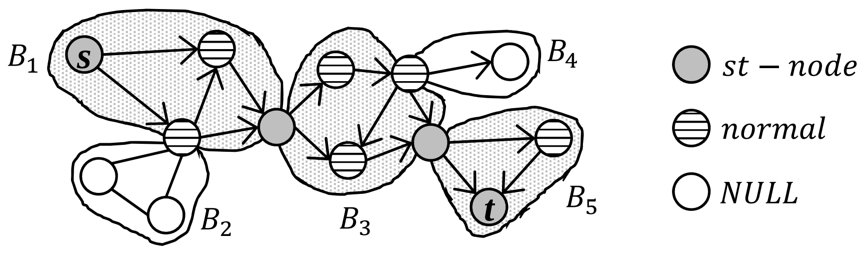


Figure 7: Process classification in algorithm *CLSP*

For the classification, algorithm *CLSP* uses a DFS tree rooted at s, the $s \overset{*}{-} t$ path and articulation points. Now we define a specific process r'.

Definition 7. *(Process r') Let T be a DFS tree rooted at s in N and assume that u is an articulation point. If u is removed from N, N is divided into two or more connected components. Each connected subgraph N' which does not include s has a subtree T' of T rooted at a new process: a child process of u in T. Process r' is a process which becomes the root process of a subtree T' when a articulation point u is removed.*

Simply, process r' is the child process of u in T.

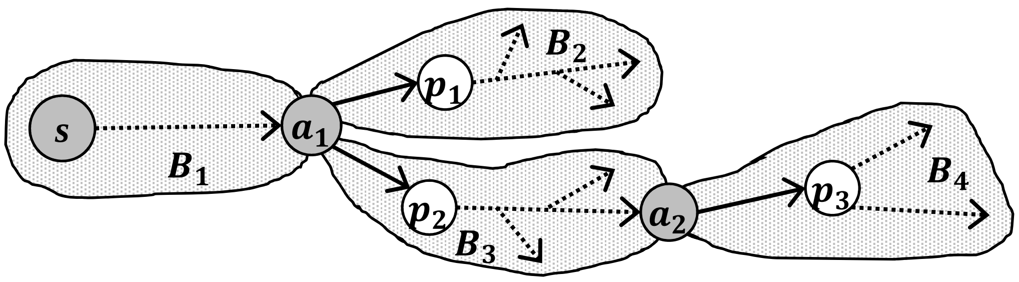


Figure 8: Process r'

Figure 8 illustrates an example of the process r'. If articulation point a_1 is removed from the network, p_1 and p_2 become r'. If articulation point a_2 is removed, p_3 becomes r'. The spanning tree rooted at r' is separated from the connected component that includes s if the articulation point is removed. Hence, the articulation point (the parent process of r') is necessary to reach s from r'. r' becomes the root process of the subtree of the DFS tree spanning over the biconnected block except for parent (articulation point) of r', thus, a path in the subtree starting at r' forms a part of the $s \overset{*}{-} t$ path if r' appears in the $s \overset{*}{-} t$ path. To check whether a process is r', we use the variable *Lowlink* in the algorithm for finding articulation points (the step 2 of our algorithm) and the traversal order (DFS preorder) of the DFS tree. $Lowlink_i$ of each process p_i stores the minimum value of the traversal orders among the neighbors of p_i's descendants (including p_i). To find *Lowlink* distributedly, we define it as follows.

Definition 8. *(Lowlink) Let r be the root of the DFS tree T. Lowlink of each process stores the following value, where $dfnum$ is the traversal order.*

- *Root process r*
  - $Lowlink_r \Leftarrow 0$
- *Non-root process p_i*
  - $min\_adj\_Lowlink \Leftarrow min\{\infty, Lowlink_j | p_j \in N_i, dfnum_j > dfnum_i\}$
  - $min\_adj\_dfnum \Leftarrow min\{dfnum_j | p_j \in N_i\}$
  - $Lowlink_i \Leftarrow min\{min\_adj\_Lowlink, min\_adj\_dfnum\}$

When *Lowlink* of a non-root process stores the traversal order of its parent, the parent process becomes an articulation point. Therefore, a process whose *Lowlink* stores the traversal order of its parent process becomes r'.

Each process can determine what type among the three types it belongs to using the above information. Algorithm *CLSP* can refer the variables of the three previous algorithms (step 1 to 3) due to the fair composition.

3.5.1 Overview of algorithm *CLSP*

Algorithm *CLSP* classifies each node into one of the three types, and each type of the node fulfills the following conditions.

- **(st-node)** A source or sink node of a transport net in the biconnected block that has an edge in the $s \overset{*}{-} t$ path. Removing the st-node (except s and t) causes the disconnection of N, thus st-node is an articulation point. When an articulation point has r' as its child process in the $s \overset{*}{-} t$ path, it becomes st-node.
- **(normal)** A normal process is a node except for the st-nodes that is included in a biconnected block including an edge of the $s \overset{*}{-} t$ path.
- **(NULL)** A NULL process is a node in a biconnected block that includes no edge of the $s \overset{*}{-} t$ path. Hence, NULL processes are r' which is not in the $s \overset{*}{-} t$ path and its all descendant nodes.

We introduce algorithm *CLSP* for classifying each node into the three types in the following subsection.

3.5.2 Variables of each process

In this subsection, we introduce variables which are maintained by each process.

- A variable which is written to a register.
  - $nodetype_i \in \{stnode, normal, NULL\}$: The type of p_i.
- Constant variables of p_i.
  - N_i: The set of p_i's neighbor processes.
  - $children_i$: The set of p_i's child processes.
  - par_i: The parent process of p_i.
  - $dfnum_i$: The traversal order of p_i (of the DFS tree).
  - low_i: *Lowlink* of p_i.

Algorithm 2 Algorithm $CLSP$ for classifying of nodes

```
1:  struct $x$ { $nodetype \in \{stnode, normal, NULL\}$ }
2:  while forever do
3:      for every $p_k \in N_i$ do $x[k] = read(R_{ki})$
4:      if $p_i$ is $s \vee p_i$ is $t$ then
5:          $nodetype_i \Leftarrow stnode$
6:      else if $(sep\_tree\_root(i) \wedge \neg sink\_exists_i) \vee x[par_i].nodetype_{par_i} = NULL$ then
7:          $nodetype_i \Leftarrow NULL$
8:      else if $art_i \wedge (\exists p_j \in children_i, sep\_tree\_root(j) \wedge sink\_exists_j)$ then
9:          $nodetype_i \Leftarrow stnode$
10:     else
11:         $nodetype_i \Leftarrow normal$
12:     end if
13:     for every $p_k \in N_i$ do $write(R_{ik}, nodetype_i)$
14: end while
```

– $sink\_exists_i$: A boolean variable to denote whether p_i is on the $s \overset{*}{-} t$ path.

– art_i: A boolean variable to denote whether p_i is an articulation point or not.

Variables $children_i$, par_i, $dfnum_i$, and low_i are adequately set in steps 1[9] and 2[8], and $sink\_exists_i$ is adequately set in algorithm $FSTP$ introduced in the previous section. Algorithm $CLSP$ can refer these variables because of the fair composition. The value of art_i can be determined from the result of the algorithm of step 2[8].

Each process uses the following predicate.

- $sep\_tree\_root(i) \equiv (low_i = dfnum_{par_i})$

  True when *Lowlink* of p_i is the same as the traversal order of its parent. Note that when *true*, p_i becomes r'.

Algorithm 2 shows an implementation of $FSTP$ using the above variables and predicate.

3.5.3 Proof of correctness

In this subsection, we prove the correctness of algorithm $CLSP$. Before the proof, we introduce some definitions.

Definition 9. *(Each node's type in a consistent configuration of algorithm CLSP)*

1. *$nodetype_i = NULL$, if p_i never appears in any $s - t$ path.*

2. *$nodetype_i = stnode$, if p_i appears in every $s - t$ path.*

3. *$nodetype_i = normal$, all other processes.*

Definition 10. *(A consistent configuration of algorithm CLSP) A configuration of N is consistent for CLSP when N fulfills the following conditions:*

1. *Each $nodetype_i$ and each register $R_{ij}(\forall j \in N_i)$'s nodetype have the same values.*

2. *Each $nodetype_i$ satisfies Definition 9.*

Lemma 2. *Lowlink of p_i and the traversal order of the parent of p_i are the same if and only if p_i becomes r',*

Proof. **(Proof of necessary condition)**: Let p_i be any process other than the source node s and p_a be the parent process of p_i. If the traversal order of p_a and *Lowlink* of p_i are the same, p_a becomes an articulation point. This implies that p_i and its descendant processes can not connect

with the process with a smaller traversal order than p_a. Therefore, when p_a is removed from T, N is partitioned into the spanning tree rooted at p_i and the others. In this case, p_i becomes r' from Definition 7.
(Proof of sufficient condition): Assume p_i becomes r' of Definition 7 and p_u is the parent process of p_i and an articulation point of N. Because of Definition 7, if p_u is removed from T, N is partitioned into two or more connected components one of which consists of nodes appearing in the subtree of T (say T') rooted at p_i. No process in T' connects with the processes out of T' except of p_u. Thus p_u has a smaller traversal order than any processes in T'. Therefore, *Lowlink* of p_i is the same as the traversal order of p_u due to Definition 8 (*Lowlink*). □

Lemma 3. *Each process which never appears in any $s - t$ is classified into NULL.*

Proof. If process p_i never appears in any $s - t$, p_i does not appear in the $s \overset{*}{-} t$ path neither. In addition, to reach s from p_i, the nearest articulation point which is the ancestor process of p_i has to be visited. This implies that p_i is r' of the articulation point or its descendant process.

1. When p_i is r': p_i is r' and is not in the $s \overset{*}{-} t$ path, thus $sep\_tree\_root(i) \wedge \neg sink\_exists_i$ becomes *true*. $nodetype_i$ becomes NULL by lines 6 and 7 of Algorithm 2.

2. When p_i is r''s descendant: Because p_i is not in any $s - t$ path, r' is not in the $s \overset{*}{-} t$ path. This makes r' NULL (refer the above case 1.), and its child processes become NULL. Therefore all descendant processes of r' become NULL.

□

Lemma 4. *All processes in the biconnected block which includes an edge of the $s \overset{*}{-} t$ path are classified into st-node or normal.*

Proof. We prove the lemma by contradiction. Assume a process p_i appears in some $s - t$ path, and becomes NULL. By lines 6 and 7 of Algorithm 2, p_i becomes a NULL process when it is r' not in the $s \overset{*}{-} t$ path or its parent process is NULL.

1. Case that p_i is r' which is not in the $s \overset{*}{-} t$ path: The subtree of the spanning tree rooted at p_i does not include t. This implies that any path from p_i to t must include the articulation point which is an ancestor of p_i. The path from p_i to s must include the same articulation point. Therefore, the same process appears twice in any $s - t$ in which p_i appears, this contradicts the assumption.

2. Case that parent node of p_i is NULL: Let p_j be the parent node of p_i. p_j executes the same algorithm, thus the condition of the classification is also the same. If p_j is r' which is not in the $s \overset{*}{-} t$ path (refer the above case 1.), this contradicts the assumption due to the case 1. Otherwise, there must be a process which is the same as the case 1 in p_j's ancestors because the number of processes is finite.

□

Lemma 5. *Each process which is included in every $s - t$ path is classified into st-node.*

Proof. If p_i is s or t, $nodetype_i$ becomes st-node due to lines 4 and 5 of Algorithm 2. Thus we consider only the case that p_i is neither s nor t.

Because p_i is included in every $s - t$ path, p_i also appears in the $s \overset{*}{-} t$ path. If p_i is removed from the network N, t is separated from s, hence p_i is an articulation point. Therefore, p_i has the child process r', and becomes st-node due to lines 8 and 9 of Algorithm 2. □

Lemma 6. *Assume that all processes execute an atomic action of algorithm CLSP at least once. On the DFS tree rooted at s, after both Algorithm FART for finding articulation points and Algorithm FSTP for finding the $s \overset{*}{-} t$ path converge, even when algorithm CLSP is executed from any configuration, all processes converge to a consistent configuration (Definition 10) within finite time.*

Proof. Due to the assumption, each process executes an atomic action of algorithm $CLSP$ at least once, therefore, its local variable *sink_exists* and each register's *sink_exists* written by the process have the same value. Each st-node process (will converge to st-node) determines its type (st-node) with refering the following information: articulation points, the $s \overset{*}{-} t$ path and r' due to lines 4, 5, 8 and 9. And its type remains unchanged and satisfies Lemma 5. Each NULL process which is r' and the r' is not included in the $s \overset{*}{-} t$ path determines its type with referring invariant information, thus its type remains unchanged. And NULL process whose parent is NULL eventually determines its type and converges by lines 6 and 7 of Algorithm 2 (Lemma 3). All st-node and NULL processes determine their type and converge, each normal process can determine its type and satisfy Lemma 4. From the above convergences, each process is classified within finite time and satisfies Lemma 15, that is algorithm $CLSP$ can reach a consistent configuration. □

The following theorem holds by Lemma 6.

Theorem 2. *Algorithm CLSP is a self-stabilizing algorithm which classifies each process into one of the three types: st-node, normal and NULL.*

3.6 St-ordering algorithm $stORDER$(step 5)

Assume that the DFS tree rooted at s is constructed, the $s \overset{*}{-} t$ path and all articulation points are found and all processes are classified into the three types (st-node, normal, NULL) in $N = (P, L)$ (or $G = (V, E)$). We use an st-ordering algorithm introduced in [2] which can be executed only on a biconnected network. However we partition the network into biconnected blocks in steps 1 to 4 of our proposed algorithm, therefore we can adopt the st-ordering algorithm in each biconnected block[2]. An st-ordering algorithm uses the following information: a parent process in the DFS tree, a set of descendant processes, a traversal order and *Lowlink*. These all information can be obtained in steps 1 to 4, thus the st-ordering algorithm can be correctly executed in our proposed algorithm using the fair-composition.

Note that an articulation point which is classified into st-node is responsible for both a source node and a sink node. Figure 9 illustrates an example of an articulation point which is an st-node. In this case, p_1 and p_2 consider α as a sink node but α is considered as a source node by p_3 and p_4. Thus, each st-node maintains the variables of both a source node and a sink node, and also operates as both of them in the st-ordering algorithm. Each neighbor process of an st-node determines whether it is a source or a sink node using *Lowlink* and a traversal order. In Fig. 9, *Lowlink*s of p_1 and p_2 are lower than the traversal order of process α, and *Lowlink*s of p_3 and p_4 are the same as the traversal order of α. Moreover, each NULL process does not refer its neighbor processes' information and each neighbor process of a NULL process does not refer its information. As a result, an st-ordering to construct a transport net is archived at each biconnected block.

3.7 Algorithm $CTRN$ for constructing a transport net on each block (step 6)

In this subsection, we introduce a self-stabilizing algorithm for constructing a transport net (for step 6) using st-ordering decided in step 5. As we introduced in the Sections 1 and 2, a transport net can be easily constructed when st-order is decided on each node. We introduce algorithm $CTRN$ for constructing a transport net on each biconnected block, a maximal (1,1)-DAMG can be constructed when the algorithm $CTRN$ converges.

Algorithm $CTRN$ is executed on each biconnected block, and each node in the biconnected block maintains its own st-order. Note that each articulation point in $s \overset{*}{-} t$ is included two biconnected blocks and maintains two st-order, one is for a source node (in a biconnected block), another is

[2]The st-ordering algorithm assumes that the number of the processes in the network is given, but st-ordering can be achieved even the number of the processes is unknown. In [2], each node calculate its own st-number based on its parent node's one in the DFS tree rooted at the source node of which st-number is 1. The st-number of the sink node eventually becomes the same number as the total number of nodes n regardless of assigning n to the sink node.

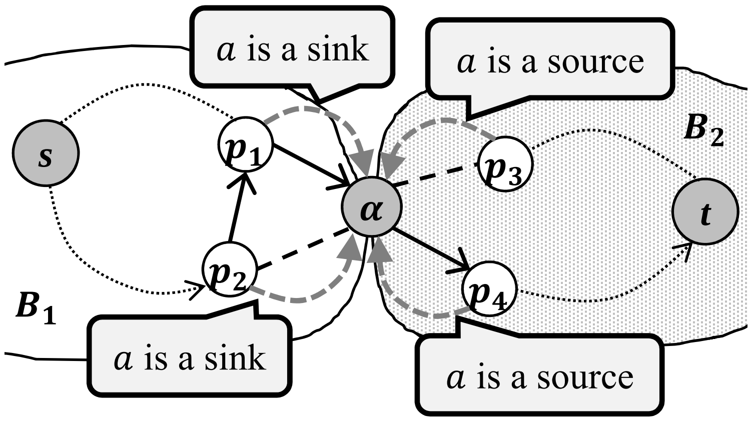


Figure 9: An articulation point which is responsible for both a source and a sink nodes

for a sink node (in a biconnected block). Each st-order is decided when the algorithm for step 5 converges, and it can be referred by each node due to a fair composition.

3.7.1 Overview of algorithm $CTRN$

We show an outline of algorithm $CTRN$ as follows.

1. An articulation point : Use its sink st-number for processes in the same block, and its source st-number (fixed at 1) for processes in the other block. Assignment directions to each edge is the same as follows (the other process).

2. The other processes : If its own st-number is $\perp$, it makes all edges (or arcs) into (undirected) edges. If not, check all neighbor node's st-number. If there is an st-number which is larger than its own st-number, changes a corresponding edge into an outgoing arc. Otherwise, changes a corresponding edge into an incoming arc. If a neighbor node's st-number is $\perp$, remains a corresponding edge undirected (or changes an arc to an edge).

3.7.2 Variables of each process

In this subsection, we introduce local variables which are maintained by each process.

- A variable which is written to a register.
  - $d_{ik} \in \{IN, OUT, NULL\}$: A variable stores the direction from p_i to p_k (p_k is a neighbor process of p_i). If there is no direction between p_i and p_k, $d_{ik} = NULL$.
- Constant variables of p_i.
  - N_i: A set of p_i's neighbor processes.
  - ST_i: An st-number of p_i. If p_i is an articulation point, ST_i stores its st-number as a (virtual) sink node.
  - art_i: A boolean variable to denote whether p_i is an articulation point or not.

Algorithm 3 shows an implementation of $CTRN$ using the variables introduced above.

Algorithm 3 Algorithm *CTRN* for constructing a transport net.

```
1: **struct** $x$ { $ST \in \mathbb{N}$ }
2: **while** every $k \in N_i$ **do**
3:     $x[k] = read(R_{ki})$
4:     **if** $art_i = true \wedge k$ is not in the same block **then**
5:         $ST_t = 1$                                        ▷ $p_i$ is a (virtual) source node
6:     **else**
7:         $ST_t = ST_i$
8:     **end if**
9:     **if** $ST_t = \perp \vee ST_k = \perp$ **then**
10:        $d_{ik} \Leftarrow NULL$
11:    **else if** $ST_t < ST_k$ **then**
12:        $d_{ik} \Leftarrow OUT$
13:    **else**
14:        $d_{ik} \Leftarrow IN$
15:    **end if**
16: **end while**
```

4 Summary

In this paper, we defined a new network structure named maximal (σ, τ)-DAMG and we proposed a self-stabilizing algorithm for constructing a maximal (1,1)-DAMG in an arbitrary connected network. We also presented the correctness proof of our proposed algorithm. However, our algorithm consists of several self-stabilizing algorithms using the fair composition, and its convergence time becomes much longer when the scale of the network becomes larger even if the topology of the network is nearly unchanged. We consider the analysis of the time complexity to converge of our algorithm as a future work. Moreover we are finding the way to simplify our algorithm.

We had already obtained a self-stabilizing algorithm for constructing a maximal (1,2)-DAMG from any arbitrary connected network, however we can not introduce it in this paper due to the lack of space.

Designing the generalized algorithm for constructing a maximal (σ, τ)-DAMG for any σ and τ is a future work. In a maximal (σ, τ)-DAMG, we assume that each source node has a directed path to at least one sink node. However, we can also consider a maximal (σ, τ)-DAMG in which every source node in S is reachable to every sink node in T. We call this a maximal *strong-connected* (σ, τ)-DAMG, and consider it as one of the future works. Finally, finding the necessary and sufficient condition for existing consistent a maximal (strong-connected) (σ, τ)-DAMG is also an important future work.

References

[1] Rohan F. M. Aranha and C. Pandu Rangan. An efficient distributed algorithm for st-numbering the vertices of A biconnected graph. *J. UCS*, 1(9):633–650, 1995.

[2] Pranay Chaudhuri and Hussein Thompson. A self-stabilizing algorithm for the st-order problem. *IJPEDS*, 23(3):219–234, 2008.

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

[4] J.E. Burns. Self-stabilizing rings without daemons. *Technical Report GIT-ICS-87/36*, Georgia Tech, 1987.

[5] Shlomi Dolev. *Self-Stabilization.* MIT Press, 2000.

[6] J. Ebert. st-ordering the vertices of biconnected graphs. *Computing*, 30(1):19–33, 1983.

[7] Mehmet Hakan Karaata and Pranay Chaudhuri. A dynamic self-stabilizing algorithm for constructing a transport net. *Computing*, 68(2):143–161, 2002.

[8] Haruka Ohno and Yoshiaki Katayama. A self-stabilizing algorithm for finding articulation points using a dfs tree with message complexity O(log n) (in Japanese). In *Tokai-Section Joint Conference on Electrical, Electronics, Information and Related Engineering, September 8–9*, 2014.

[9] Keisuke Okamoto and Yoshiaki Katayama. A self-stabilizing algorithm for constructing a dfs tree with message complexity O(log n) (in Japanese). In *WTCS '13, Proceedings of the 11th Workshop on Theoretical Computer Science, Karatsu, Saga, September 11–13*, pages 118–128, 2013.

[10] Ji-Cherng Lin and Ming-Yi Chiu. Short correctness proofs for two self-stabilizing algorithms under the distributed daemon model. *Discrete Applied Mathematics*, 157, 140–148, 2009.