The Spanning-tree Algorithm and Generating a Membrane Structure

Šárka Vavrečková

Silesian University in Opava, Bezručovo nám. 13, Opava, Czech Republic,
sarka.vavreckova@fpf.slu.cz

Abstract: Many articles on membrane systems seek to minimize the number of membranes. But if these systems were to be used for practical purposes, we may need a system with a large number of membranes – the proper arrangement of this structure would be very difficult. In this paper, we propose a procedure inspired by the spanning-tree algorithm to help optimize a complex structure of membranes, independently of system rules.

1 Introduction

Membrane computing is a framework of parallel distributed processing introduced by Gheorghe Păun in 1998. Research in this area is ongoing and many different types of membrane systems have emerged since then. Information is available at [12, 14, 15, 16], or the bibliography at [18].

The intent of Membrane computing is to abstract computing using hierarchies of membranes in living cells. Since 2000, the term P Systems has been used for mathematical models of membrane systems.

The minimum spanning tree (MST) for a given structure (network) of nodes is a substructure of the form of a tree with optimal paths leading from nodes to the root node, while maintaining the established rules for relationships between nodes, even when the structure is changed (adding new nodes, removing them or changing properties of connections). There are many applications of the MST algorithm in various areas – network design, network optimization, maximum bottleneck paths, image processing, grouping objects into clusters of similar objects, approximation of some NP-hard problems (e.g. traveling salesperson problem), etc., see [1].

There are several algorithms to find the MST: Kruskal's algorithm, Prim's algorithm, Boruvka's algoritm, the algorithm in the STP protocol (Spanning-Tree Protocol). Some of them are discused in [3, 7].

We will deal with the algorithm implemented in STP, because, in addition to mathematical representation, it is well described in the form of code, although for the purposes of computer networks, and standardized. The MST algorithm implemented in the STP protocol maintains a loop-free logical topology with network devices (Ethernet switches or bridges), keeping backup physical connections that are immediately available in the event of a change in logical topology. STP was standardized as IEEE 802.1D, the current revision is IEEE 802.1D-2004, reaffirmed in 2011 [2, 17].

The MST algorithm is one of the graph algorithms, and it is one of the most known greedy algorithms. Greedy algorithms are heuristic problem-solving algorithms searching for locally optimal choices with the intent to discover a global optimum. More information is available at [6, 7].

The paper [8] describes the implementation of Dijkstra's algorithm (finding the shortest path in a graph) using P system, which is closely related to the spanning-tree, but a membrane system is a tool there, not a target.

A dynamic structure of membranes (P Systems with active membranes) is discussed in [13] or [4], but the presented structure was able to dissolve membranes and to divide a membrane into two successor membranes. Each membrane has its polarization (electrical charge with values $\{-,0,+\}$), and the polarization of a membrane can be changed as well. All changes of membranes are made using rules.

In this paper, we work with dynamics of membrane structure in a slightly different manner. Changes in the membrane structure are not built in the rules of the membrane system, but they are meta-rules used to pre-prepare the structure of the system, which will then work according to its own rules.

2 Analogy between Membrane Structure and Tree Structure

The basis of membrane systems is a membrane structure inspired by the structure of a biological cell. A membrane can contain certain objects and/or other (nested) membranes. Thus, there may be a parent-child relationship between the membranes. Objects can be manipulated using rules, in some membrane systems there are also rules for manipulating membranes.

The membrane structure is strictly hierarchical, it can be represented by a sequence of nested brackets or displayed by a tree in which the parent-child relationship corresponds to the inner-outer membrane relationship. One membrane (the skin) is the main one (it contains all the others), the remaining membranes always have one parental membrane in which they are contained. Analogously, the tree has one main (root) node, all other nodes have exactly one parent node. The representation using a tree intuitively leads to the possibility of using graph algorithms designed for graphs in the form of a tree.

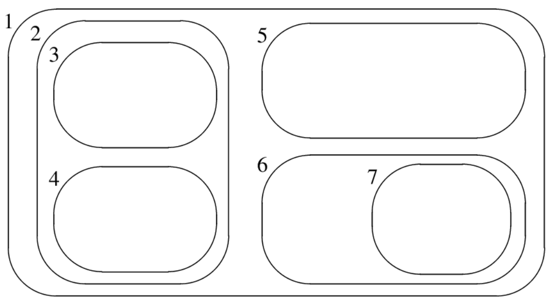

Figure 1: Example of Membrane Structure

The membrane structure can be represented graphically using Venn diagrams, brackets, or a tree of nodes representing membranes. The membrane structure presented in Figure 1 by Venn diagram is represented by the string $[\,[\,[\,]_3\,[\,]_4\,]_2\,[\,]_5\,[\,[\,]_7\,]_6\,]_1$, and by the tree of nodes shown in Figure 2.

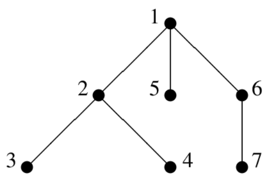

Figure 2: Tree of nodes

The membranes are identified by their labels. The membrane structure presented in Figures 1 and 2 uses simple numbers as labels.

There are many articles and procedures that deal with the rules and objects in the membrane structure, but the membranes themselves are supposed to be "somehow designed", their structure is not usually specifically addressed. It is not necessary for a few membranes; however, if this structure is to be large and very complex (which can happen, for example, in a structure for simulating the operation of many mutually hierarchically arranged systems), we may encounter problems in the design of such a structure. In the case of a complicated structure, it may happen that we do not correctly express suitable relationships between membranes, analogously in the tree we do not correctly express parent-child relationships between nodes.

A tool we use for the design of a membrane structure can allow input in the form of nested brackets or in graphical form. In brackets, we can get lost in small tens of membranes, the graphic input is lucid only till a certain extent of the network: with tens to hundreds of membranes, we can get lost here as well. Or defining parenthood in membranes can be manually challenging for larger structures.

In this paper, we propose an optimization process that can help organize a very large membrane structure. The process remains laborious, but increases the probability of obtaining a better optimized structure.

In local computer networks (LAN) we work with network devices of the type switch. The role of the switch is to receive a data unit called a frame from one port and forward it to another port, behind which the target device for the given frame is located.

In general, we find it useful when there are redundant paths in a network. But if we create redundant paths in the network of switches, we also create loops, which have a negative impact on network operation (broadcast frames intended for all nodes in the network flow by the network in multiple copies, we call it "broadcast storm"[1]). The solution is to leave redundant paths at the physical level (cables), but remove them at the logical level (communication) by blocking data communication on a given connection. In case that the active path stops to forward frames, one of the logicaly blocked paths whose physical connection exists will be activated.

In the case of switches, the entire process is directed by the STP protocol, which implements the spanning-tree algorithm. The purpose of this algorithm is to negotiate its position in the hierarchy when connecting the switch to the network and to keep the hierarchy in optimal condition according to the general rules and set priorities when changing the network structure. Each node has assigned its own priority (in a parent-child relationship, the parent will have a smaller priority number). The algorithm does not depend on the number of nodes, it runs in parallel at all nodes and affects the structure of the entire network at the same time.

When designing a large membrane structure, for each membrane we usually have an idea of how deep in the structure it should be roughly, and also which membranes are considered parental for it. We want to choose one of the potential parents for the given membrane in the optimization process as the parent membrane, and thus actually include the given membrane into the structure (both the membrane and the tree structure). We can solve the choice of the main membrane or the root node of the tree for the whole system similarly.

3 Preliminaries

3.1 Membrane Structure and P Systems

We assume the reader to be familiar with the basics of the formal language theory and membrane computing. For further details we refer to [10] and [16].

The definition of P Systems follows, but only some parts of the definition are important for the purposes of this article. Different types of P Systems have quite different definitions, and the mechanism we present is applicable to various types.

Each membrane has its *region*: the space delimited by the given membrane, all contained objects and subordinate membranes (with their contained objects) are situated in this region. The region of a membrane corresponds to the subtree in the tree representation of the structure.

[1]The definition of broadcast storm can be found e.g. at `https://www.techopedia.com/definition/6270/broadcast-storm`.

Definition 1 ([15]). *Let H be a set of labels. A P System of a degree m, $m \geq 1$, is a construct*

$$\Pi = (V, T, C, \mu, w_1, \ldots, w_m, (R_1, \rho_1), \ldots, (R_m, \rho_m))$$

where:

(i) *V is a nonempty alphabet, its elements are called objects,*

(ii) *T is an output alphabet, $T \subseteq V$,*

(iii) *C is a set of catalysts, $C \subseteq V - T$,*

(iv) *μ is a membrane structure consisting of m membranes, the membranes are labeled by the elements of H,*

(v) *w_i, $1 \leq i \leq m$, are strings representing multisets over V associated with the region of the $i - th$ membrane in μ,*

(vi) *R_i, $1 \leq i \leq m$, are finite sets of evolution rules associated with the region of the $i - th$ membrane in μ; an evolution rule is a pair (u, v), also written $u \rightarrow v$, where*

- *u is a string over V,*
- *$v = v'$ or $v = v'\delta$, where v' is a string over $\{a_{here}, a_{out}, a_{in_j} \mid a \in V,\ 1 \leq j \leq m\}$, and δ is a special symbol $\notin V$ representing dissolution of membrane,*

(vii) *ρ_i, $1 \leq i \leq m$, is a partial order relation over R_i specifying the priorities of the rules in R_i.*

Details and examples can be found in [15].

3.2 The Spanning-tree Algorithm

For the purposes of this article, we will simplify the original spanning-tree algorithm and list only those parts of it that we will use for operations with a membrane structure. The full algorithm can be found in [9] (multiple pages) and certainly [2]. The presented algorithms are created loosely according to [9, 2, 5].

As mentioned above, the spanning-tree algorithm is the parallel algorithm working on nodes of a network, e.g. on interconnected network switches. The goal is to transform the network into a tree with one root node. The process of transformation is called *convergence*.

The algorithm was originally intended for network devices called bridge, so the term "bridge" also appears in the following text.

Assume a network of nodes (bridges, switches, membranes,...), interconnected in some manner. There can be more than one path between some nodes, the structure does not yet have the form of a tree, the convergence process is in progress.

Every node keeps the following information:

- *BID* (Bridge ID) consisting of two parts:
  - *priority* of the node,
  - unique *deviceID*, e.g. MAC address of a switch,

  BID can be represented by the vector
  (priority ; deviceID)
  or by concatenation of these numbers (i.e. a number), the priority is more significant item than deviceID,
- *rootID* is BID of the root node, so every node knows the root,
- *root path cost* (sum of costs of all links on the way to the root),
- each port linked to a neighbor has its *portID* consisting of two parts: port priority and port number (counted from 1), these two properties can be represented by a vector or concatenation similarly to BID,
- each port has three additional features: role, current status, and cost of link.

The *port status* is either forwarding (active) or blocking (the full algorithm defines more than two port states, also depending on the protocol version).

The *port role* is one of the following:

- root port – the active port through which the path to the root node leads (leading to the parent node),
- designated port – the active port leading to one of the child nodes (or leading to another subtree, blocked by the neighbor),
- alternate port – non-active (blocked) port.

The original algorithm distinguishes two types of blocked ports (alternate or backup).

Algorithm 1: Basic Structures and Functions

```
node  :  BID (priority, label),
         rootID (priority, label),
         RPcost,
         rootPortID, //portID of the root port
         //ports are indexed by portID:
         ports[] (role, state, cost,
            neighborBID, neighborPortID);

function node.init()
begin
    //Neighbors' BIDs and their portIDs are
      determined later from obtained BPDUs.
    //Setting port costs,...
    self.IAmNewRoot();  //Defined in Algorithm 2.
end

//No BPDU came from the neighbor for a hold
  time: the link or the neighbor is inoperative.
function node.noRespFromNeigh(recPortID)
begin
    if (recPortID == self.rootPortID) then
        //The root port waiting in vain for a BPDU.
          The root has become unavailable, maybe
          I am the new root.
        self.IAmNewRoot();
end
```

The *port cost* (or link cost, it should be the same value on the both sides of the link) is used to calculate the optimal path over network. The faster the port/link, the lower this number (and better link).

The representation of a node and its initialization can be found in Algorithm 1.

The nodes communicate with each other using *BPDU messages* (Bridge Protocol Data Unit). BPDU is valid only for one link and the corresponding ports, between two neighbors; it is not resent to another link, the neighbor generates the own BPDUs for its links. Every BPDU contains, inter alia, the following information:

- BID of the sender,
- rootID (BID of the node that is being considered root by the sender),
- root path cost (cost of the path from the sender to the root),
- portID and cost of the port the BPDU is sent over,
- other information.

The neighbors know each other through the BPDUs, including BIDs and portIDs, and the root path cost is calculated by adding the cost of the link to the root path cost info obtained from the neighbor.

There are several types of BPDUs: *Proposal BPDU* is sent downwards to inform its child nodes about the root node, *Agreement BPDU* is sent upwards to agree with the proposed root setting (see Algorithm 4), and others which are not necessary for our purposes.

The BPDUs are also used as a keep-alive mechanism between neighbor nodes and the neighbors inform each other about their parameters (BIDs, port numbers, etc.).

Presume a situation where a new node is added to the network. This node first assumes that it is the root, and starts sending BPDUs with BID = rootID. All its ports have the role DP (designated port). Its neighbors receive the BPDU with this information and compare the reported rootID with the stored value. There are two possibilities:

1. the reported rootID is greater than the stored rootID – the new node does not become the root,
2. the reported rootID is less than the stored rootID – the new node becomes the root.

The function node.IAmNewRoot() in Algorithm 2 illustrates this situation.

Example 1. *The numbers deviceID are usually MAC addresses. For demonstration reasons, we will use only "small" numbers for the both priorities and deviceIDs.*

If a node has the stored rootID = (32 ; 56) and another node is reporting rootID = (31 ; 82), the reported node becomes the new root (the lower number of priority means precedence). But if another node later reports rootID = (31 ; 95), the reported node will not become the root (its priority is the same as the root priority, but its deviceID is greater, which means no precedence).

Algorithm 2: Auxiliary functions

```
function node.changeRootPort(newRootPortID,
  newPortRole)
begin
    // rootPortID==0 would mean that I am the root
    if (self.rootPortID != 0) then
        self.ports[self.rootPortID].state = blocking;
        self.ports[self.rootPortID].role =
          newPortRole;
    end
    self.rootPortID = newRootPortID;
    self.ports[newRootPortID].state = blocking;
    self.ports[newRootPortID].role = rootPort;
end

// Obtained info about new root or root path cost
   changed, do synchronization downwards.
function node.makeSync(sender)
begin
    // Only connected/used ports are processed:
    foreach (port in self.ports) do
        port.state = blocking;
    self.ports[self.rootPortID].state = forwarding;
    self.ports[self.rootPortID].sendBPDU_agree();
    foreach (port in self.ports) do
        if ((port.role = = designatedPort)
        or (port.role = = alternatePort)) then
          port.sendBPDU_proposal() ;
end

function node.IAmNewRoot()
begin
    self.rootID = self.BID;
    self.RPcost = 0;
    self.rootPortID = 0;
    foreach (port in self.ports) do
        port.role = designatedPort;
        port.state = blocking;
        port.sendBPDU_proposal();
    end
end
```

If a node obtains a BPDU with the declared rootID less (i.e. better) than its stored rootID, it updates the own structures and sends the BPDUs to all neighbors (except of the original sender), so these neighbors learn the change, subsequently their neighbors learn the change, etc.

The root node has the root path cost = 0 for all its ports, so the newly connected node (considering to be root) sends BPDUs with this value to its neighbors. Each receiving neighbor adds the cost of the receiving port/link to the obtained root path cost information (the first command of the function node.getBPDU_proposal() in Algorithm 3). The next step depends on which port the BPDU came over. The first part of Algorithm 3 (the first "if" command block) il-

Algorithm 3: Getting BPDU: Proposal

```
// BPDU of the type "Proposal" received on the port "recPortID" from the given "sender" node.
function node.getBPDU_proposal(recPortID, sender)
begin
    refRPcost = sender.RPcost + self.ports[recPortID].cost;  // Root path cost from sender + cost between us
    self.ports[recPortID].neighborBID = sender.BID;  // Discovering neighbor's BID
    self.ports[redPortID].neighborPortID = sender.portID;
    if (self.ports[recPortID].role = = rootPort) then  // Info originated from the root port is trusted.
        if ((sender.rootID < self.rootID) or ((sender.rootID = = self.rootID) and (refRPcost != self.RPcost))) then
            // Path to the root has changed, but the direction remains.
            self.ports[self.rootPortID].state = blocking;  // blocking before any change of ports
            self.rootID = sender.rootID;
            self.RPcost = refRPcost;
            self.makeSync(sender);  // including sending BPDU Agreement upwards and Proposal downwards
        else if (sender.rootID > self.BID) then  // The root has become unavailable, maybe I am the new root.
            self.IAmNewRoot()
        return;
    end
    if (sender.rootID < self.rootID) then  // New root notified, the reported rootID is better than I know:
        self.changeRootPort(recPortID, designatedPort);  // The receiving port becomes my new root port.
        self.rootID = sender.rootID;
        self.RPcost = refRPcost;
        self.makeSync(sender);
    else if (sender.rootID = = self.rootID) then
        // The same root, maybe some other change – root path cost or direction?
        if (refRPcost < self.RPcost) then  // The root path cost has changed to a better value:
            changeRootPort(recPortID, designatedPort);
            self.RPcost = refRPcost;
            self.makeSync(sender);
        else if (refRPcost > self.RPcost) then  // The root path cost has changed to a worse value:
            if (sender.BID > self.BID) then  // The sender is either under me or inside another subtree
                self.ports[recPortID].role = designatedPort;  // previously could have been alternatePort
                self.makeSync(sender);
            else  // means: (sender.BID < self.BID), the sender is above me or inside another subtree
                self.ports[recPortID].state = blocking;
                self.ports[recPortID].role = alternatePort;
            end
        else  // The same root and root path cost, but from different direction, choosing more eligible parent:
            if (sender.BID > self.ports[rootPortID].neighborBID) then  // The present parent is better, no change.
                self.makeSync(sender);
            else if ((sender.BID < self.ports[rootPortID].neighborBID)
              or ((sender.BID = = self.ports[rootPortID].neighborBID)
              and (self.ports[recPortID].neighborPortID < self.ports[rootPortID].neighborPortID))) then
                // The potential new parent has better BID, changing the root port:
                foreach (port in self.ports) do  if (port.role = = alternatePort) then  port.role = designatedPort ;
                self.changeRootPort(recPortID, alternatePort);
                foreach (port in self.ports) do  if (port.role = = designatedPort) then
                    port.state = blocking;
                    port.sendBPDU_proposal();
                end
            end
        end
    end
end
```

lustrates the case for the root port of the node, while the next code represents the case for designated and alternate ports.

The node considers the reported rootID, root path cost and other parameters to decide the best direction to the root and the corresponding port roles. If the BPDU with the correct rootID and the same lowest root path cost comes to more than one port (and therefore it is not possible to select the root port according to the above procedure), the lower sender BID decides. If the sending nodes have the same BID (e.g. when two or more parallel links lead between two nodes), the portIDs of the sender's port and the neighbor's port leading to current root are compared and the port with the neighbor's lower portID is selected as the root port.

The configured ports are in the blocking state during determining port roles.

The both the reported rootID and the root path cost together flood in all directions from the root node: first to its neighbors (they count the cost of the link leading to the root plus zero), then to their neighbors (the value increases again, they add the cost of the link to the obtained value), etc. Each node gets the necessary information gradually, the topology may change several times during the convergence; when a node detects a root change, it communicates information about the new root to its ports (except the root port) during the synchronization process (the function node.makeSync(sender), see Algorithm 2).

Whenever there is a change in the structure, in the priorities of nodes or ports, etc., this procedure starts again and the whole network gradually converges. If a node finds out about the change:

- it evaluates the change reported in the BPDU proposal,
- if the node agrees, it changes its configuration and sends back the BPDU agreement,
- it blocks the ports to other neighbors and sends them the BPDU proposal.

In case that a neighbor agrees, it sends back the BPDU agreement (the port leading to this neighbor is activated with the role "designated"). If it does not agree, it sends back the own proposal in the BPDU proposal.

Algorithm 4: Getting BPDU: Agreement

```
// BPDU "Agreement" received on the port
   "recPortID", my proposal is accepted by the
   neighbor
function node.getBPDU_agree(recPortID)
begin
    self.ports[recPortID].role = designatedPort;
    self.ports[recPortID].state = forwarding;
end
```

4 Optimization of Membrane Structure

The algorithm can be run over an existing graph from which we want to create a tree, or individual elements (membranes) can be added gradually.

Consider a set of membranes, each of them has

- contained objects, properties, functions, rules, etc., according to the type of membrane system,
- its own identifier (label) and priority (i.e. BID); the priority is added as metainformation for the purposes of the algorithm,
- the identifier and priority of the root membrane (skin; i.e. rootID); it will probably be appropriate to determine the skin membrane in advance and assign it the lowest BID number directly,
- ports to several possible neighbors in the membrane structure (potential parents and children) – links (relations) to them, for each port we need its portID, the neighbor BID and the portID for the link on the neighbor's side, each such link is evaluated by cost as metainformation.

There may be left default values for added properties, or we can set them manually to impact the resulting structure.

To make the procedure as illustrative as possible, we use the representation of the membrane structure in the form of a tree. The procedure described in Algorithms 1, 2, 3, and 4, is intended for network devices (however, very simplified), but we can use this simplified version for membranes and just modify the terms.

Algorithm 5 is created by modifying Algorithm 1, the remaining algorithms can be modified in a similar way. The port priorities are not necessary (we need just port labels as portIDs), because we do not assume multiple connections between two membranes.

Example 2. *Let us demonstrate the above outlined procedure. For the sake of clarity, the individual membranes are created sequentially, but we would achieve the same result even if all membranes were created in parallel or when the order was changed, only the convergence process would pass in different timing. The following initial structure is given:*

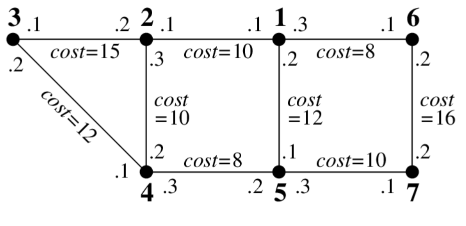

Figure 3: Initial structure for Example 2

There are 7 membranes with connections and link costs, the ports are marked in a smaller font and with a point, e.g. the membrane no. 3 has the ports 3.1 (leading to the membrane no. 2) and 3.2 (leading to the membrane no. 4). All

Algorithm 5: Base for Membranes

```
membrane: BID (priority, label),
          rootID (priority, label), RPcost,
          rootPortID,
          ports[] (role, state, cost,
             neighborBID, neighborPortID);

function membrane.create(BID, neighs[], costs[])
begin
    self.BID = BID;
    int c = neighs.count;
    if c > 0 then
        for (int i = 1; (i <= MaxPorts) and (i <= c);
         i++) do
            self.ports[i].neighborBID = neighs[i];
            self.ports[i].cost = costs[i];
            ... ;  // according to implementation
        end
    self.IAmNewRoot();
end

function membrane.noRespFromNeigh(recPortID)
begin
    if (recPortID = = self.rootPortID) then
        self.IAmNewRoot();
end
```

membranes have the same priority: only labels are used to make decisions when comparing BIDs. This structure is not in the form of tree, so we apply the transformation.

Assume that the structure is formed gradually, for example from lower labels till larger labels (but the order is not important). The steps are as follows:

- *the membrane no. 1 is created, it is the root (the skin membrane), all its ports have the role DP,*
- *the membrane no. 2 is created and connected to the membrane no. 1; considered to be the root (sending Proposal), but receives the message from no. 1 with the lower (i.e. better) rootID (so sending back Agreement), the root is no. 1, the port gets the role RP,*
- *the membrane no. 3 is created, connected to no. 2; considered to be the root (sending Propossal), but receives back the message from no. 2 with the lower rootID (sending back Agreement), the root is no. 1, the port 3.1 gets the role RP,*
- *the membrane no. 4 is created, connected to no. 2 and no. 3; considered to be the root (sending Proposal), but receives back the message from no. 2 and 3 with the lower rootID (sending back Agreement), the root is no. 1; the root path cost through no. 2 is* $10+10=20$*, through no. 3 is* $10+15+12=37$*, so the port 4.2 gets the role RP, the port 4.1 gets the role AP (because sender.BID < self.BID), the port 3.1 of the membrane no. 3 goes to the role DP,*
- *the membrane no. 5 is created, connected to no. 1 and 4; considered to be the root (sending Proposal), but receives back the message from the neighbors with the lower rootID (sending back Agreement), the root is no. 1; the root path cost from no. 1 is better* ($0+12$)*, so the port 5.1 gets the role RP, the port 5.2 gets th role AP, the opposite port 4.3 gets the role DP,*
- *the membrane no. 4 found an alternative path to the root (no. 1) through no. 5 with the same root path cost; but the neighbor no. 2 has better BID, so the root port remains,*

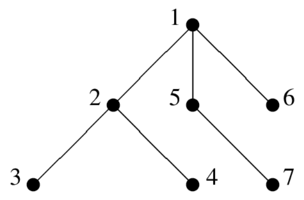

Figure 4: Modified structure (tree) for Example 2

- *the membrane no. 6 is created, connected to no. 1, after exchanging messages the root is no. 1, the port 6.1 gets the role RP,*
- *the membrane no. 7 is created, connected to no. 5 and 6, after exchanging messages the root is no 1, the port 7.1 gets the role RP (the root path cost through no. 5 is better), the opposite port 5.3 gets the role DP, the port 7.2 gets the role AP and the opposite port 6.2 gets the role DP.*

Example 3. *Setting various membrane priorities and different link costs causes the topology change. Let us take the previsous example and set the priority of the membrane no. 2 to 100, the others to 200, and change the cost of the link between membranes no. 6 and 7 to the value 8.*

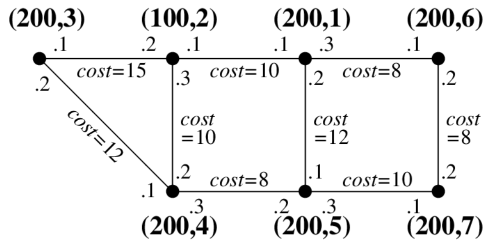

Figure 5: Initial structure for Example 3

As we can see (Figure 6), the structure has changed quite a lot.

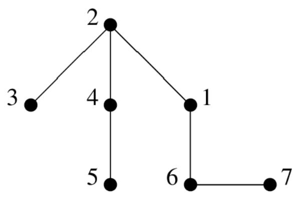

Figure 6: Modified structure (tree) for Example 3

5 Conclusions

The aim of this work is not to design a comprehensive tool for simulation of P systems, there is not enough space for that. The goal is only to design a procedure for generating suitable input, e.g. for P-Lingua type languages[11].

Membrane systems can be used to simulate complex (real) systems with many relations. Real systems are being changed over time (not only by deleting existing membranes) and the proposed procedure offers a way of representing mutual relations that can be dynamically adapted.

The two examples show the consequences for the change in the membrane structure caused by the intervention in only two parameters – the priority of one membrane and the cost of one link (relation).

There is only one problem left to solve: if there are several potential child membranes before using the algorithm and only some of them are selected by the algorithm, or the parent-child relationship may change, then some rules a_{in_j} may lose relevance. This must be taken into account when designing evolution rules, or the presented algorithm can be enriched with the possibility of dynamic adaption of this type of evolution rules; of course, if the side effects of this modification will be acceptable.

The work is supported by The Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science – LQ1602.

References

[1] Applications of Minimum Spanning Tree Problem [online]. Geeks for Geeks (2018). URL: `https://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/` (Accessed August 14, 2020)

[2] 802.1D-2004 – IEEE Standard for Local and metropolitan area networks: Media Access Control (MAC) Bridges [online]. IEEE (2011). URL: `https://ieeexplore.ieee.org/document/1309630` (Accessed June 4, 2020)

[3] Bazlamaçcı, C.F., Hindi, K.S.: Minimum-weight spanning tree algorithms: A survey and empirical study. Computers & Operations Researche, vol. 28, issue 8, ISSN 0305-0548, pp. 767–785 (2001), online available at: `http://www.sciencedirect.com/science/article/pii/S0305054800000071`

[4] Calude, C., Păun, Gh.: Computing with Cells and Atoms. Taylor and Francis, London, 2000 (Chapter 3: “Computing with Membranes”).

[5] ComputerNetworkingNotes: STP – Spanning Tree Protocol Explained With Examples [online]. Networking Tutorials (2020). Online available at `https://www.computernetworkingnotes.com/ccna-study-guide/stp-spanning-tree-protocol-explained-with-examples.html` (Accessed June 4, 2020)

[6] Curtis, S.A.: The classification of greedy algorithms. Elsevier, 2003. Also available at `https://core.ac.uk/download/pdf/82042073.pdf` (Accessed June 4, 2020)

[7] Greenberg, H.J.: Greedy Algorithms for Minimum Spanning Tree [online]. Univeristy of Colorado (1998), online available at `http://glossary.computing.society.informs.org/notes/spanningtree.pdf` (Accessed June 4, 2020)

[8] Guo, P., Quan, Ch., Ye, L.: UPSimulator: A general P system simulator. Knowledge-Based Systems, vol. 170, 2019, pp. 20–25. ISSN: 0950-7051.

[9] Hewlett Packard Enterprise: Calculation process of the STP algorithm [online]. HPE FlexNetwork LAN Switching Configuration Guide (2017), online available at `https://techhub.hpe.com/eginfolib/networking/docs/switches/7500/5200-1938a_l2-lan_cg/content/495503520.htm` (Accessed June 11, 2020)

[10] Hopcroft, J.E., and Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, 1979.

[11] P-Lingua, Main Page [online]. Online available at `http://www.p-lingua.org/wiki/index.php/Main_Page` (Accessed August 11, 2020)

[12] Păun, Gh.: Computing with membranes. J. Comput. Syst. Sci. **61**(1), 108–143 (2000). Turku Center for Computer Science-TUCS report 208, November 1998.

[13] Păun, Gh.: Computing with membranes – A Variant: P Systems with Polarized Membranes. Intern. J. of Foundations of Computer Science, vol. 11, issue 1 (2000), pp. 167–182.

[14] Păun, Gh.: Membrane Computing: An Introduction. Springer, Heidelberg (2002).

[15] Păun, Gh., Rozenberg, G.: A Guide to Membrane Computing. Theor. Comp. Science, vol. 287, issue 1, pp. 73–100 (2002).

[16] Păun, Gh., Rozenberg, A., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, New York (2010).

[17] US Patent “Using Spanning Tree Protocol (STP) to Enhance Layer-2 Network Topology Maps”, patent number US 8,045,488 B2, 2011. Also available at `https://patents.google.com/patent/US8045488B2/en`. (Accessed June 4, 2020)

[18] The P Systems Website: `http://psystems.eu` (Accessed June 4, 2020)