Journal of Membrane Computing (2023) 5:108–115
https://doi.org/10.1007/s41965-023-00120-1

RESEARCH PAPER

Networks of splicing processors: simulations between topologies

José Angel Sanchez Martín[1] · Victor Mitrana[2] · Mihaela Păun[3,4]

Received: 30 November 2022 / Accepted: 3 March 2023 / Published online: 6 April 2023

Abstract
Networks of splicing processors are one of the theoretical computational models that take inspiration from nature to efficiently solve problems that our current computational knowledge is not able to. One of the issues restricting/hindering is practical implementation is the arbitrariness of the underlying graph, since our computational systems usually conform to a predefined topology. We propose simulations of networks of splicing processors having arbitrary underlying graphs by networks whose underlying graphs are of a predefined topology: complete, star, and grid graphs. We show that all of these simulations are time efficient in the meaning that they preserve the time complexity of the original network: each computational step in that network is simulated by a fixed number of computational steps in the new topologic networks. Moreover, these simulations do not modify the order of magnitude of the network size.

Keywords Splicing processor · Network of splicing processors · Underlying graph · Simulation.

1 Introduction

The formal operation of *splicing* on strings has been introduced in [5] as an abstraction of the biological phenomenon of DNA recombination under the effect of restriction and ligases enzymes. The biological phenomenon is illustrated in Fig. 1. We give here a few informal explanations. Two DNA molecules (the blue and the red ones) are cut by a restriction enzyme (in this case the enzyme is EcoRI). This process yields fragments with Watson–Crick complementary tails called "sticky ends". These sticky ends may join again leading to the recombination of DNA. To fix the new combination, a DNA enzyme called ligase seals the gaps after the sticky ends are joint.

We follow [15] with the formal definition of splicing as an operation on pairs of strings. First, we need to define what a splicing rule is: a quadruple of strings specifying the subsequences in the two strings where the strings are cut. Therefore, a splicing rule is intended to abstract the restriction enzymes and its subsequences indicate the sites where the enzymes cut. Different computational models based on the iteration of this operation may be defined. Thus, a generating splicing system initiates a computation starting from a given finite set of strings (axioms) and iteratively applying splicing rules, from a given finite set of such rules, producing eventually a language. This computational model was introduced in [5]; further on, the model and its variants have intensively been investigated. Splicing operation, as a formal operation on words and languages, has been vividly studied for more than two decades. There have been published a lot of papers as well as several books containing chapters devoted to this topic. We mention here just a few of them [6, 9, 16], containing extensive chapters about splicing, as well as [7, 8], containing chapters that intend to discuss various applications. There are two types of splicing systems: *generating systems*, which generates a language by iteratively applying

✉ Victor Mitrana
victor.mitrana@upm.es

José Angel Sanchez Martín
josanc16@ucm.es

Mihaela Păun
mihaela.paun@incdsb.ro

1 Department of Software Engineering and Artificial Intelligence, Universidad Complutense de Madrid, Calle del Prof. José García Santesmases, 9, 28040 Madrid, Spain

2 Department of Information Systems, Universidad Politecnica de Madrid, Calle Alan Turing s/n, 28031 Madrid, Spain

3 Bioinformatics Department, National Institute for R &D for Biological Sciences, 296 Independenţei Bd., Bucharest 060031, Romania

4 Faculty of Administration and Business, University of Bucharest, Bucharest, Romania

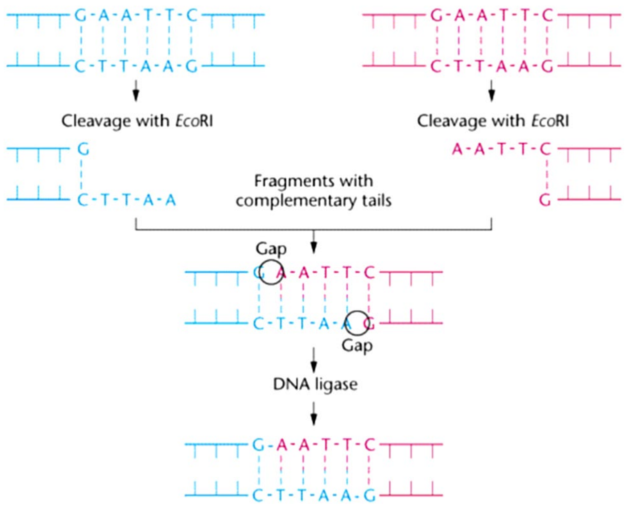


Fig. 1 Splicing operation (Klug and Cummings 1997)

splicing rules to the strings obtained starting from a finite set of strings, and *accepting system*, which starts out with just one initial string and a finite set of axioms and an iterative splicing as above is initiated. The computation halts when at least one string from a predefined set is obtained. The input string is accepted as soon as the system halts. The accepting splicing system has been introduced by Mitrana et al. in [13], while different variants have been studied in [1, 4, 14], etc.

In [10] a highly parallel and distributed computational model based on the splicing operation was introduced: *network of splicing processors (NSP)*. This model consists in an undirected graph whose nodes host a splicing processor. A splicing processor consists in a finite set of splicing rules, a finite set of strings (axioms) and four sets of symbols, such that two of them define the input filter while the other two define the output filter. A computation in a network of splicing processors (NEP, for short) is a sequence of splicing and communication steps which alternate with each other. In a splicing step, each processor applies, in parallel, the splicing rules it contains to all the strings existing at that moment in the processor. Note that we assume that each string appearing in a processor at some moment, appears actually in an unlimited number of identical copies such that different copies may be rewritten by different splicing rules. In a communication step, all the strings existing in the network nodes are simultaneously are expelled from their nodes, provided that they can pass the output filters of the nodes. In the same communication step, arbitrary large number of copies of each string expelled from one node (sender) enter all the nodes (receivers) connected to the sender, provided that the string can pass the input filters of the receivers. The computation halts as soon as a predefined node, called *Halt*, contains at least a string.

Several variants of NSP have been considered so far, most of them being computationally complete, see, e.g., [2, 3, 10–12]. These networks have an ad hoc underlying graph structure. By different reasons like: possible implementations, uniformity, comparisons, etc., it would be useful to have networks with a fixed and well known topology as: complete graph, star, grid, etc. This is actually the aim of this work: to investigate the possibility of transforming a given NSP into an equivalent NSP with an underlying graph of such a predefined structure. We are interested not only in the construction of these networks but also in comparing the computational time and size of the constructed networks with those of the original ones.

2 Basic definitions

In this section we introduce the main concepts and notations that will be used in the sequel. For those notions not defined here we refer to [17].

An *alphabet* is a finite and nonempty set of symbols. The cardinality of a finite set A is written $card(A)$. Any finite sequence of symbols from an alphabet V is called *string* over V. The set of all strings over V is denoted by V^* and the empty string is denoted by ε. The length of a string x is denoted by $|x|$ while $alph(x)$ denotes the minimal alphabet W such that $x \in W^*$. A language over the alphabet V is a set $L \subseteq V^*$.

We give now the formal definition of the splicing operation following [15]. A *splicing rule* over a finite alphabet V is a quadruple of strings of the form $[(u_1, u_2);(v_1, v_2)]$ such that u_1, u_2, v_1, and v_2 are in V^*. For a splicing rule $r = [(u_1, u_2);(v_1, v_2)]$ and for $x, y, z \in V^*$, we say that r produces z from x and y (denoted by $(x, y) \vdash_r z$) if there exist some $x_1, x_2, y_1, y_2 \in V^*$ such that $x = x_1u_1u_2x_2$, $y = y_1v_1v_2y_2$, and $z = x_1u_1v_2y_2$. For a language L over V and a set of splicing rules R we define

$$\sigma_R(L) = \{z \in V^* \mid \exists u, v \in L, \exists r \in R \text{ such that } (u, v) \vdash_r z\}.$$

A short discussion is in order here. As one can see, the splicing rule defined above is a 1-splicing rule in the sense of [6]. However, in the rest of the paper we do not make any difference between the two strings a splicing rule is applied to, therefore we may say that the rules are actually 2-splicing rules .

Let V be an alphabet; we now define two predicates, one with strong conditions (s) and another with weak restrictions (w), for a string $z \in V^+$ and two disjoint subsets P, F of V as follows:

$$\varphi^{(s)}(z;P, F) \equiv P \subseteq alph(z) \ \wedge \ F \cap alph(z) = \emptyset$$
$$\varphi^{(w)}(z;P, F) \equiv alph(z) \cap P \neq \emptyset \ \wedge \ F \cap alph(z) = \emptyset.$$

In the definition of these predicates, the set P is a set of *permitting symbols* while the set F is a set of *forbidding symbols*. Informally, both conditions require that no forbidding symbol occurs in z. As one can see, the former condition is stronger than the second one since it requires that all permitting symbols are present in z, while the latter requires that at least one permitting symbol appears in z.

These predicates are extended to a language $L \subseteq V^*$ by

$$\varphi^{\beta}(L,P,F) = \{w \in L \mid \varphi^{\beta}(w;P,F)\},$$

with $\beta \in \{(s),(w)\}$.

A *splicing processor* over an alphabet V is a 6-tuple (S, A, PI, FI, PO, FO), where:

- S is a finite set of splicing rules over V.
- A is a finite set of *auxiliary strings* over V. These auxiliary strings are to be used, together with the existing strings, in the splicing steps of the processors. Auxiliary strings are available at any moment.
- $PI, FI \subseteq V$ are the sets of permitting and forbidding symbols, respectively, which form the *input* filter of the processor.
- $PO, FO \subseteq V$ are the sets of permitting and forbidding symbols, respectively, which form the *output* filter of the processor.

The set of splicing processors over V is denoted by SP_V.

A *network of splicing processors* is a 9-tuple $\Gamma = (V, U, \langle, \rangle, G, \mathcal{N}, \alpha, \underline{In}, \underline{Halt})$, where:

- V and U are the input and network alphabet, respectively, $V \subseteq U$, and $\langle, \rangle \in U \setminus V$ are two special symbols.
- $G = (X_G, E_G)$ is an undirected graph without loops with the set of nodes X_G and the set of edges E_G. Each edge is given in the form of a binary set. G is called the *underlying graph* of the network.
- $\mathcal{N} : X_G \longrightarrow SP_U$ is a mapping, which associates with each node $x \in X_G$ the splicing processor $\mathcal{N}(x) = (S_x, A_x, PI_x, FI_x, PO_x, FO_x)$.
- $\alpha : X_G \longrightarrow \{(s),(w)\}$ defines the type of the filters of a node.
- $\underline{In}, \underline{Halt} \in X_G$ are the *input* and the *halting* node of Γ, respectively.

The *size* of an NSP Γ is defined as the number of nodes of the graph, i.e., $card(X_G)$. A *configuration* of an NSP Γ is a mapping $C : X_G \to 2^{U^*}$, which associates a set of strings with every node of the graph. Although a configuration is a multiset of strings, each one appearing in an arbitrary number of copies, for the sake of simplicity, we work with the support of this multiset. A configuration can be seen as the sets of strings, except the auxiliary ones, which are present in the nodes at some moment. For a string $w \in V^*$, we define the initial configuration of Γ on w by $C_0^{(w)}(\underline{In}) = \{\langle w \rangle\}$ and $C_0^{(w)}(x) = \emptyset$ for all other $x \in X_G$.

A configuration is followed by another configuration either by a splicing step or by a communication step. A configuration C' follows a configuration C by a splicing step if each component $C'(x)$, for some node x, is the result of applying all the splicing rules in the set S_x that can be applied to the strings in the set in $C(x)$ together with those in A_x. Formally, configuration C' follows the configuration C by a splicing step, written as $C \Rightarrow C'$, iff for all $x \in X_G$, the following holds:

$$C'(x) = \sigma_{S_x}(C(x) \cup A_x).$$

In a communication step, the following actions take place simultaneously for every node x:

(i) all the strings that can pass the output filter of a node are sent out of that node;
(ii) all the strings that left their nodes enter all the nodes connected to their original ones, provided that they can pass the input filter of the receiving nodes.

Note that, according to this definition, those strings that are sent out of a node and cannot pass the input filter of any node are lost. Formally, a configuration C' follows a configuration C by a communication step (we write $C' \models C$) iff for all $x \in X_G$

$$C'(x) = (C(x) - \varphi^{\alpha(x)}(C(x), PO_x, FO_x)) \cup \bigcup_{\{x,y\} \in E_G} (\varphi^{\alpha(y)}(C(y), PO_y, FO_y) \cap \varphi^{\alpha(x)}(C(y), PI_x, FI_x))$$

holds. For an NSP Γ, a computation on an input string w is defined as a sequence of configurations $C_0^{(w)}, C_1^{(w)}, C_2^{(w)}, ...,$ where $C_0^{(w)}$ is the initial configuration of Γ on w, $C_{2i}^{(w)} \Rightarrow C_{2i+1}^{(w)}$ and $C_{2i+1}^{(w)} \models C_{2i+2}^{(w)}$, for all $i \geq 0$. A computation on an input string *w halts* if there exists $k \geq 1$ such that $C_k^{(w)}(\underline{Halt})$ is non-empty. Such a computation is called an *accepting computation*. As the halting node is used just for ending the computation, we shall consider that $S_{\underline{Halt}} = A_{\underline{Halt}} = \emptyset$. Furthermore, because as soon as a string enters $\underline{Halt}$, the computation halts and no string goes out, we may also consider that $PO_{\underline{Halt}} = FO_{\underline{Halt}} = \emptyset$.

The language accepted by Γ is defined as

$$L(\Gamma) = \{z \in V^* \mid \text{the computation of } \Gamma \text{ on} z \text{is an accepting computation}\}.$$

Given an NSP Γ with the input alphabet V, we define the following computational complexity measure. The *time complexity* of the finite computation $C_0^{(x)}, C_1^{(x)}, C_2^{(x)}, \ldots C_m^{(x)}$ of Γ

on $x \in V^*$ is denoted by $Time_\Gamma(x)$ and equals m. The time complexity of Γ is the partial function from $\mathbf{N}$ to $\mathbf{N}$,

$$Time_\Gamma(n) = \max\{Time_\Gamma(x) \mid x \in L(\Gamma), |x| = n\}.$$

3 Complexity preserving simulations

3.1 Simulating arbitrary NSP by complete NSP

Theorem 1 *For every NSP Γ one can construct a complete NSP Γ' such that the following conditions are satisfied*:

1. $L(\Gamma) = L(\Gamma')$.
2. $Time_{\Gamma'}(n) \in \mathcal{O}(Time_\Gamma(n))$.
3. $size(\Gamma') = size(\Gamma) + 2$.

Proof Let $\Gamma = (V, U, <, >, G, \mathcal{N}, \beta, \underline{In}, \underline{Halt})$ be a NSP with the underlying graph $G = (X_G, E_G)$ and $X_G = \{x_1, x_2, \dots, x_n\}$ for some $n \geq 2$; $x_1 \equiv \underline{In}$ and $x_n \equiv \underline{Halt}$. We construct the NSP $\Gamma' = (V', U', <, >, G', \mathcal{N}', \beta', \underline{In'}, \underline{Halt'})$, where

$$V' = V, \quad U' = U \cup T \cup \{\#\}, \quad T = \{t_i \mid 0 \leq i \leq n\},$$

G' is the complete graph K_{n+2} represented in Fig. 2, with the nodes $\{\underline{In'}, x_{comp}, x_1^s, x_2^s, \cdots, x_{n-1}^s, \underline{Halt'}\}$ defined as follows:

- node $\underline{In'}$:

  $S = \{[(a, >);(\#, > t_0)] \mid a \in U\}$, $A = \{\# > t_0\}$,
  $PI = U \setminus (T \cup \{\#\})$, $FI = \{\#\} \cup T$,
  $PO = U \setminus \{\#\}$, $FO = \{\#\}$,
  $\beta' = (w)$

- node x_{comp}:

  $S = \{[(\varepsilon, t_j);(\#, t_i)](\mid 1 \leq j \neq i \leq n) \wedge (\{x_i, x_j\} \in E_G)\} \cup \{[(\varepsilon, t_0);(\#, t_1)]\}$, $A = \{\#t_i \mid 1 \leq i \leq n\}$,
  $PI = U \setminus \{\#\}$, $FI = \{\#\}$,
  $PO = U \setminus \{\#\}$, $FO = \{\#\}$,
  $\beta' = (w)$

- node x_i^s, $1 \leq i \leq n-1$:

  $S = S_{x_i}$, $A = \{zt_i \mid z \in A_{x_i}\}$,
  $PI = PI_{x_i}$, $FI = FI_{x_i} \cup T \setminus \{t_i\}$,
  $PO = PO_{x_i}$, $FO = FO_{x_i}$,
  $\beta' = \beta(x_i)$

- node $\underline{Halt'}$:

  $S = \emptyset$, $A = \emptyset$,
  $PI = PI_{\underline{Halt}}$, $FI = FI_{\underline{Halt}}$,
  $PO = \emptyset$, $FO = \emptyset$,
  $\beta = \beta(\underline{Halt})$.

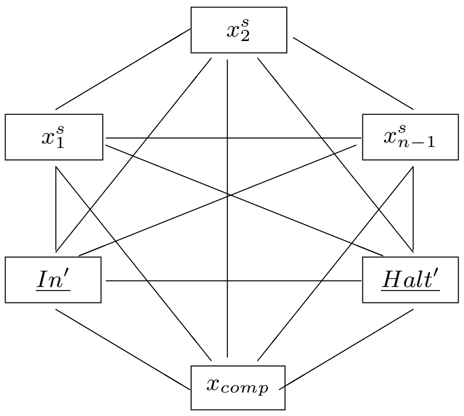


Fig. 2 The underlying graph of Γ'

We now analyze a computation of Γ' on the input string $< w >$. In the input node $\underline{In'}$, the symbol t_0 is attached to the end of the string. Next, the symbol t_0 is replaced by t_1 in the node x_{comp}. When it goes out, it can only enter x_1^s and the simulation of a computation in Γ starts. Thus, the string $< w > t_1$ lies in x_1^s, while the string $< w >$ is found in x_1, the input node of Γ. More generally, we may assume that a string zt_i is found in a node $x_i^s \in \Gamma'$ if and only if the corresponding string z lies in $x_i \in \Gamma$, for all $1 \leq i \leq n-1$. Note that the strings can never return to $\underline{In'}$ because of the input filter of this node.

Let x_i be a splicing node, where a rule $[(a, b); (u, v)]$ is applied to w yielding w' and w''. Then, the same rule is applied in x_i^s and strings of the form $w't_i$ and $w''t_i$ are produced. Indeed, since all the strings in $A_{x_i^s}$ and any string entering x_i^s have the symbol t_i at the end, the splicing rule will always yield strings keeping the character t_i as the last one. Since both the node x_i and the node x_i^s have the same output filters and the produced strings only differ in this last character t_i, it follows that a string can only leave x_i^s if and only if the original counterpart can exit x_i. Once it leaves, the string returns to x_{comp} and the character t_i is replaced with t_j characters in different copies, provided that $\{x_i, x_j\} \in E_G$. Each of the copies is sent to the corresponding connected node x_j and the process described above restarts. It immediately follows that $L(\Gamma') = L(\Gamma)$.

It is easy to notice that $Time_{\Gamma'}(w) = 2Time_\Gamma(w)$ for every $w \in L(\Gamma)$, hence the second statement is proved.

Finally, this construction needs two more nodes, therefore $size(\Gamma') = size(\Gamma) + 2$. □

3.2 Simulating arbitrary NSP by star NSP

Theorem 2 *For every NSP Γ one can construct a star NSP Γ' such that the following conditions are satisfied*:

1. $L(\Gamma) = L(\Gamma')$.
2. $Time_{\Gamma'}(n) \in \mathcal{O}(Time_{\Gamma}(n))$.
3. $size(\Gamma') = size(\Gamma) + 2$.

Proof The simulation is identical to the one for complete graphs. The node x_{comp} is set as the center of the star network, while all the other nodes defined in the previous proof are connected to it, as shown in Fig. 3.

Clearly, each computation in Γ' goes as in the previous construction, hence all the statement of the theorem follow. □

3.3 Simulating arbitrary NSP by grid NSP

Theorem 3 *For every NSP Γ one can construct a grid NSP Γ' such that the following conditions are satisfied*:

1. $L(\Gamma) = L(\Gamma')$.
2. $Time_{\Gamma'}(n) \in \mathcal{O}(Time_{\Gamma}(n))$.
3. $size(\Gamma') = 3size(\Gamma) + 3$.

Proof Let $\Gamma = (V, U, <, >, G, \mathcal{N}, \beta, \underline{In}, \underline{Halt})$ be a NSP with the underlying graph $G = (X_G, E_G)$ and $X_G = \{x_1, x_2, \dots, x_n\}$ for some $n \geq 2$; $x_1 \equiv \underline{In}$ and $x_n \equiv \underline{Halt}$. We construct the NSP $\Gamma' = (V, U', <, >, G', \mathcal{N}', \beta', \underline{In'}, \underline{Halt'})$, where

$$\begin{aligned}
U' &= U \cup T \cup \{\#\},\\
T &= \{t_i, t_i', t_i^1, t_i^2 \mid 1 \leq i \leq n\},\\
G' &= (X_{G'}, E_{G'}),\\
X_{G'} &= \{\underline{In'}, \underline{Halt'}, D\} \cup \{x_i^s \mid 2 \leq i \leq n-1\}\\
&\quad \cup \{x_i^{comp}, x_i^{connect} \mid 1 \leq i \leq n\}.
\end{aligned}$$

The underlying graph of the network Γ' is the grid graph with width 3 and height $n+1$ from Fig. 4 below and its nodes are defined as follows:

- node $\underline{In'}$:

  $S = \{[(a, >);(\#, > t_1)] \mid a \in U\}$, $A = \{\# > t_1\}$,
  $PI = U' \setminus (\{\#\} \cup T)$, $FI = \{\#\} \cup T$,
  $PO = U' \setminus \{\#\}$, $FO = \{\#\}$,
  $\beta' = (w)$

- nodes x_i^s, $1 \leq i \leq n-1$:

  $S = S_{x_i}$, $A = \{zt_i \mid z \in A_{x_i}\}$,
  $PI = PI_{x_i}$, $FI = FI_{x_i} \cup T \setminus \{t_i\} \cup \{\#\}$,
  $PO = PO_{x_i}$, $FO = FO_{x_i}$,
  $\beta' = \beta(x_i)$

- nodes D, D':

  $S = \emptyset$, $A = \emptyset$,
  $PI = \emptyset$, $FI = U'$,
  $PO = \emptyset$, $FO = U'$,
  $\beta' = (s)$

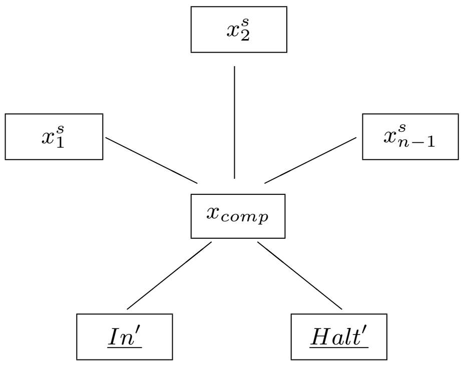


Fig. 3 The underlying graph of Γ'

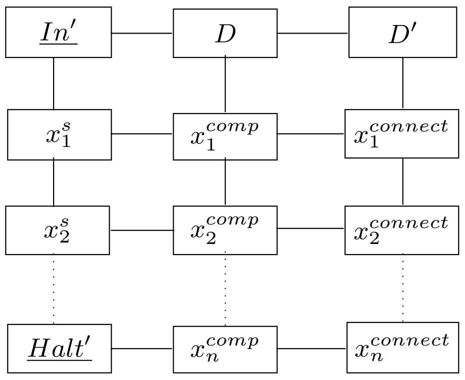


Fig. 4 The underlying graph of Γ'

- nodes $x_i^{comp}, 1 \le i \le n$:

$$S = \{[(\varepsilon, t_i');(\#, t_i)]\} \cup$$

$$\begin{cases} \{[(\varepsilon, t_i);(\#, t_j^1)] \mid 1 \le i \ne j \le n \wedge \{x_i, x_j\} \in E_G\}, \\ \text{if i is an even number,} \\ \{[(\varepsilon, t_i);(\#, t_j^2)] \mid 1 \le i \ne j \le n \wedge \{x_i, x_j\} \in E_G\}, \\ \text{if i is an odd number,} \end{cases}$$

$$A = \begin{cases} \{\#t_j^1 \mid 1 \le i \ne j \le n \wedge \{x_i, x_j\} \in E_G\}, \\ \text{if i is an even number,} \\ \{\#t_j^2 \mid 1 \le i \ne j \le n \wedge \{x_i, x_j\} \in E_G\}, \\ \text{if i is an odd number,} \end{cases}$$

$PI = \{t_i, t_i'\}, \quad FI = \{\#\},$
$PO = U' \setminus \{\#\}, FO = \{\#\},$
$\beta' = (w)$

- $\underline{\text{nodes } x_i^{connect} \mid 1 \le i \le n}$:

$$\overline{S = \{[(\varepsilon, t_i^1);(\#, t_i')]\} \cup \{[(\varepsilon, t_i^2);(\#, t_i')]\} \cup}$$

$$\begin{cases} \{[(\varepsilon, t_j^1);(\#, t_j^2)] \mid 1 \le i \ne j \le n\}, \\ \text{if i is an even number,} \\ \{[(\varepsilon, t_j^2);(\#, t_j^1)] \mid 1 \le i \ne j \le n\}, \\ \text{if i is an odd number,} \end{cases}$$

$$A = \{\#t_i'\} \cup \begin{cases} \{\#t_j^2 \mid 1 \le i \ne j \le n\}, \\ \text{if i is an even number,} \\ \{\#t_j^1 \mid 1 \le i \ne j \le n\}, \\ \text{if i is an odd number,} \end{cases}$$

$$PI = \begin{cases} \{t_j^1 \mid 1 \le j \le n\}, \text{ if i is an even number,} \\ \{t_j^2 \mid 1 \le j \le n\}, \text{ if i is an odd number,} \end{cases}$$

$FI = \{\#\}, PO = U', FO = \emptyset, \beta = (w)$

We now analyze a computation of Γ' on the input string $< w >$. In the input node $\underline{In'}$, the symbol t_1 is attached at the end. Next, the string enters x_1^s and the simulation of a computation in Γ starts. Thus, the string $< w > t_1$ lies in x_1^s while the string $< w >$ is found in x_1, the input node of Γ. More generally, we may assume that a string zt_i is found in a node $x_i^s \in \Gamma'$ if and only if the corresponding string z lies in $x_i \in \Gamma$. Note that the strings cannot longer return to $\underline{In'}$ because of its FI filter. Note that the node $\underline{In'}$ and the nodes D and D' will not accept any string from now on because of their PI filters. Consequently, the first row can be disregarded for the rest of the computation.

Let x_i be a splicing node, where a rule $[(a, b); (u, v)]$ is applied to w yielding w' and w''. Then, the same rule is applied in x_i^s and strings of the form $w't_i$ and $w''t_i$ are produced. Indeed, since all the strings in $A_{x_i^s}$ and any string entering x_i^s have the symbol t_i at the end, the splicing rule will always yield strings keeping the character t_i as the last one. Since both the node x_i and the node x_i^s have the same output filters and the produced strings only differ in this last character t_i, it follows that a string can leave x_i^s if and only if the original counterpart can exit x_i. Once it leaves the node, the string can only enter the linked node x_i^{comp} and, depending on if i is an odd or an even number, the character t_i is replaced with t_j^1 or t_j^2 characters in different copies, respectively, granted that $\{x_i, x_j\} \in E_G$. Because of this last transformation, the yielded strings can only enter the node $x_i^{connect}$. At this point, a string of the form wt_j^1 or wt_j^2 continues through the column of nodes $x_i^{connect}$ until it reaches the node $x_j^{connect}$. More precisely, in $x_i^{connect}$ the symbols t_j^1 and t_j^2 are switched alternatively, forcing the string to go simultaneously to $x_{i-1}^{connect}$ and $x_{i+1}^{connect}$, provided that $i-1 \ge 1$, $i+1 \le n$. In this way, the string eventually arrives to the node $x_j^{connect}$ and either the character t_j^1 or the symbol t_j^2 is replaced with t_j' blocking the string from continuing through the column of nodes $x_i^{connect}$. Lastly, this last character is replaced by t_j in x_j^{comp} and the string enters the intended node x_j^s, granted that it meets the requirements set by the input filters of this last node. Otherwise, it is lost. Summarizing, we consider a splicing step in Γ, that produces a string z' from z in node $x_i, 1 \le i \le n$, which is further sent to $x_j, j > i$ (the case $j < i$ is analogous). These two steps (splicing and communication) are simulated in Γ' by a series of splicing steps such that the string zt_i is transformed into $z't_i$ in x_i^s, then sent, via an itinerary that starts with the node x_i^{comp}, continues with the nodes $x_i^{connect}, x_{i+1}^{connect}, \ldots, x_j^{connect}$, and finishes with the nodes x_j^{comp} and x_j^s. Therefore, the induction step is valid. From this reasoning, we infer that $L(\Gamma) = L(\Gamma')$. Following closely the explanations, we note that each splicing step in the node x_i of Γ is simulated by at most $n + 3$ splicing steps in Γ'. This is done as follows: one step in x_i^s, followed by one step in x_i^{comp}, and then at most n splicing steps in the nodes from $x_i^{connect}$ to $x_j^{connect}$. Finally, one more step is done in x_j^{comp} before the string enters x_j^s. Since the size of Γ is constant, it follows the second statement of the theorem. The third statement is immediately valid from the Fig. 4. □

4 Conclusions and further work

Motivated by possible implementations, we have investigated the possibility of transforming an NSP with an arbitrary underlying graph into an equivalent NSP (the two have the same computational power) with an underlying graph of a predefined topology. We have considered here the complete, star, and grid graphs. We have proposed constructions for these transformations such that: (i) these constructions do not increase the time complexity, and (ii) these constructions do not increase the network size by more than a constant. The protocol of communication of the networks considered here is based on some random context conditions. We would like to investigate whether or not similar constructions can be obtained for networks of polarized splicing processors, where the protocol of communication is regulated by the polarization of the nodes and a mapping that defines the polarization of data.

Acknowledgements Work supported by the National Core Program of the National Research, Development and Innovation Plan 2022-2027, funded by the Romanian Ministry of Research and Innovation, project code 23020101.

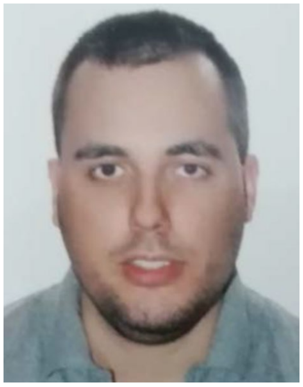
Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Arroyo, F., Castellanos, J., Dassow, J., Mitrana, V., & Sanchez-Couso, J. R. (2013). Accepting splicing systems with permitting and forbidding words. *Acta Inf., 50*, 1–14. https://doi.org/10.1007/s00236-012-0169-8
2. Bordihn, H., Mitrana, V., Păun, A., Păun, M. (2017). Networks of polarized splicing processors. In Theory and Practice of Natural Computing, TPNC 2017, Lecture Notes in Computer Science 10687, 165–177. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-319-71069-3_13
3. Bordihn, H., Mitrana, V., Negru, M. C., Păun, A., & Păun, M. (2018). Small networks of polarized splicing processors are universal. *Natural Computing, 17*, 799–809. https://doi.org/10.1007/s11047-018-9691-0
4. Castellanos, J., Mitrana, V., & Santos, E. (2011). Splicing systems: accepting versus generating. In Models of Computation in Context. CiE 2011, Lecture Notes in Computer Science. *Springer, Berlin, Heidelberg, 6735*, 41–50. https://doi.org/10.1007/978-3-642-21875-0_5
5. Head, T. (1987). Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviours. *Bull. Math. Biol., 49*, 737–759. https://doi.org/10.1007/BF02481771
6. Head, T., Păun, G., & Pixton, D. (1996). Language theory and molecular genetics: Generative mechanisms suggested by DNA recombination. *In Handbook of Formal Languages, 2*, 295–360. https://doi.org/10.1007/978-3-662-07675-0_7
7. Head, T. (2011). How the structure of DNA molecules provides tools for computation. In Biology, Computation and Linguistics. Frontiers in Artificial Intelligence and Applications vol. 228, 3–8. IOS Press. https://doi.org/10.3233/978-1-60750-762-8-3
8. Head, T. (2012). Restriction enzymes in language generation and plasmid computing In Biomolecular Information Processing: From Logic Systems to Smart Sensors and Actuators, 245–263. *Wiley Online Library*. https://doi.org/10.1002/9783527645480.CH13
9. Jonoska, N., Păun, G., Rozenberg, G. (Eds.) (2004). Aspects of Molecular Computing. Essays Dedicated to Tom Head on the Occasion of His 70th Birthday, Lecture Notes in Computer Science vol. 2950. Springer, Berlin, Heidelberg. https://doi.org/10.1007/b94864
10. Loos, R., Manea, F., & Mitrana, V. (2009). On small, reduced, and fast universal accepting networks of splicing processors. *Theoretical Computer Science, 410*, 406–416. https://doi.org/10.1016/j.tcs.2008.09.048
11. Manea, F., Martín-Vide, C., Mitrana, V. (2006). All NP-problems can be solved in polynomial time by accepting networks of splicing processors of constant size. In: DNA Computing. Lecture Notes in Computer Science, vol. 4287, 47–57. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11925903_4
12. Manea, F., Martín-Vide, C., & Mitrana, V. (2007). Accepting networks of splicing processors: complexity results. *Theoretical Computer Science, 371*, 72–82. https://doi.org/10.1016/j.tcs.2006.10.015
13. Mitrana, V., Petre, I., & Rogojin, V. (2010). Accepting splicing systems. *Theoret. Comput. Sci., 411*, 2414–2422. https://doi.org/10.1016/j.tcs.2010.03.025
14. Mitrana, V., Păun, A., & Păun, M. (2021). Non-preserving accepting splicing systems. *Joual Automata Languages Combinatorics, 26*, 109–124. https://doi.org/10.25596/jalc-2021-109
15. Păun, G. (1996). On the splicing operation. *Discrete Applied Mathematics, 70*, 57–79. https://doi.org/10.1016/0166-218X(96)00101-1
16. Păun, G., Rozenberg, G., & Salomaa, A. (1998). DNA computing: New Computing Paradigms. *Springer, Berlin, Heidelberg.*https://doi.org/10.1007/3-540-48523-6_9
17. Rozenberg, G., & Salomaa, A. (1997). Handbook of Formal Languages. *Springer, Berlin, Heidelberg.*https://doi.org/10.1007/978-3-662-07675-0



José Angel Sanchez Martín received his PhD from the Universidad Politécnica de Madrid (Spain) in September 2021 and is currently a researcher of the Department of Software and Artificial Intelligence at Complutense University of Madrid. He has authored and co-authored several articles in high-impact factor journals. His research interests include natural computing, bioinformatics, precision medicine, deep learning, and machine learning.

Victor Mitrana is a Full Professor at the Department of Information Systems of the Polytechnic University of Madrid. He has authored or co-authored more than 220 research papers published in referred journals, international academic conferences, and collective books. He has been awarded the Gheorghe Lazar Prize (1997) of the Romanian Academy, Alexander von Humboldt Fellowship (1995–1996), Ramon y Cajal Researcher (2002–2008), and is a member of the Editorial Board of seven journals and PC member of more than 40 conferences (chair for four of them). His research interests include theory of computing, algorithms and data structures, and biocomputing.

Mihaela Păun received her B.S. degree from the University of Bucharest in Computer Science in 1998, the MSc in Computer Science from the University of Western Ontario in 2000 and the PhD in Computational Analysis and Modeling and Applied Statistics from Louisiana Tech University in 2006. She is currently a Senior Researcher at the National Institute of Research and Development for Biological Sciences. Her current research interests include biostatistics and biocomputing, membrane computing, high performance computing, and environmental data analysis.