Asynchronous P Systems (Draft)

Tudor Bălănescu[1], Radu Nicolescu[2], and Huiling Wu[2]

[1] Department of Computer Science, University of Piteşti,
Târgu din Vale 1, 110040 Piteşti, Romania,
tudor_balanescu@yahoo.com

[2] Department of Computer Science, University of Auckland,
Private Bag 92019, Auckland, New Zealand,
r.nicolescu@auckland.ac.nz, hwu065@aucklanduni.ac.nz

Summary. In this paper, we propose a new approach to fully asynchronous P systems, and a matching complexity measure, both inspired from the field of distributed algorithms. We validate our approach by implementing several well-known distributed depth-first search (DFS) and breadth-first search (BFS) algorithms. Empirical results show that our P algorithms achieve a performance comparable to the standard versions.

Key words: P systems, synchronous, asynchronous, distributed, depth-first search, breadth-first search

1 Introduction

P systems is bio-inspired computational model, based on the way in which chemicals interact and cross cell membranes, introduced by Păun [20]. The essential specification of a P system includes a membrane structure, objects and rules. Cells evolve by applying rules in a non-deterministic and (potentially maximally) parallel manner. These characteristics make P systems a promising candidate as a model for distributed and parallel computing.

The traditional P system model is *synchronous*, i.e. all cells evolution is controlled by a single global clock. P systems with various *asynchronous* features have been investigated by recent research, such as Casiraghi et al. [3], Cavaliere et al. [6, 4, 5], Freund et al. [11], Gutiérrez et al. [12], Kleijn et al. [13], Pan et al. [18], Yuan et al. [24]. Here we are looking for similar but simpler definitions, closer to the definitions used in the field of distributed algorithms [14, 22], which will enable us to consider essential distributed feature, such as fairness, safety, liveness and possibly infinite executions. In our approach, algorithms are non-deterministic, not necessarily constrained to return exactly the same result.

Fully asynchronous P systems are characterized by the absence of any system clock, much less a global one; however, an outside observer may very well use a clock to time the evolutions. Our approach does *not* require any change in the *static* descriptions of P systems, only their *evolutions* differ (i.e. the underlying P engine works differently):

- Local rules execution takes *zero* time units (i.e. it occurs instantaneously).
- The message delay is unpredictable, so outgoing objects can arrive at the target cell in *any* number of time units (after being sent).

For the purpose of *time complexity*, the time unit is chosen greater than any message delay, i.e. the delay between sending and receiving a message is any real number in the closed interval $[0, 1]$.

This paper is organized as follows. Section 2 gives a definition of a simple P module, as a unified model of various P systems. Section 3 presents asynchronous P systems and discusses a standard set of time complexity measures. Section 4 and Section 5 discuss several well-known distributed DFS and BFS algorithms and propose corresponding asynchronous P system implementations. Section 6 compares the complexity of our asynchronous P system algorithms with the theoretical complexity of distributed DFS and BFS algorithms. Finally, Section 7 summarizes our work and highlights future work.

2 Preliminary

In this paper, we use *simple P modules*, an umbrella concept, which is general enough to cover several basic P system families, with *states*, *priorities*, *promoters* and *duplex* channels. For the full definition of P modules and modular compositions, we refer readers to [10].

Essentially, a simple P module is a system, $\Pi = (O, \sigma_1, \sigma_2, \dots, \sigma_n, \delta)$, where:

1. O is a finite non-empty alphabet of *objects*;
2. $\sigma_1, \dots, \sigma_n$ are cells, of the form $\sigma_i = (Q_i, s_{i,0}, w_{i,0}, R_i)$, $1 \leq i \leq n$, where:
   – Q_i is a finite set of *states*;
   – $s_{i,0} \in Q_i$ is the *initial state*;
   – $w_{i,0} \in O^*$ is the *initial multiset* of objects;
   – R_i is a finite *ordered* set of rewriting/communication *rules* of the form:
   $s\ x \rightarrow_\alpha s'\ x'\ (y)_\beta\ |_z$, where: $s, s' \in Q_i$, $x, x', y, z \in O^*$, $\alpha \in \{min, max\}$, $\beta \in \{\uparrow, \downarrow, \updownarrow\}$.
3. δ is a set of *digraph* arcs on $\{1, 2, \dots, n\}$, without reflexive arcs, representing *duplex* channels between cells.

The membrane structure is a digraph with duplex channels, so parents can send messages to children *and* children to parents. Rules are prioritized and are applied in *weak priority* order [19]. The general form of a rule, which transforms state s to state s', is $s\ x \rightarrow_\alpha s'\ x'\ (y)_{\beta_\gamma}\ |_z$. This rule consumes multiset x, and then (after all applicable rules have consumed their left-hand objects) produces multiset x', in the same cell ("*here*"). Also, it produces multiset y and sends it, by *replication* ("*repl*" mode), to all parents ("*up*"), to all children ("*down*") or to all parents and children ("*up and down*"), according to the target indicator $\beta \in \{\uparrow, \downarrow, \updownarrow\}$.

We also use a targeted sending, $\beta = \uparrow_j, \downarrow_j, \updownarrow_j$, where j is either an arc label or a cell ID. If j is an arc label, y is sent via the arc labelled j, provided that it points, respectively, up (to a parent), down (to a child) or in any direction (to either a parent or a child). If j is a cell ID of a structural neighbor, y is sent to that neighbor j, provided that it lies, respectively, up (j is a parent), down (j is a child) or in any direction (j is either a parent or a child); nothing is sent if cell j is not a structural neighbor (we do not use teleportation). More about cell IDs in a following paragraph.

$\alpha \in \{min, max\}$ describes the rewriting mode. In the *minimal* mode, an applicable rule is applied once. In the *maximal* mode, an applicable rule is used as many times as possible and all rules with the same states s and s' can be applied in the maximally parallel manner. Finally, the optional z indicates a multiset of promoters, which enable rules but are not consumed.

Note

The algorithms presented in this paper make full use of duplex channels and work regardless of specific arc orientation. Therefore, to avoid superfluous details, the structure of our sample P systems will be given as undirected graphs, with the assumption that the results will be the same, regardless of actual arc orientation.

Extensions

In this article, we use an extended version of the basic P module framework, described above. Specifically, we assume that each cell $\sigma_i \in K$ was "blessed" from factory with a unique *cell ID* symbol ι_i, which is exclusively used as an *immutable promoter*. We also allow high-level rules, with a simple form of *complex symbols* and *free variable* matching.

To explain these additional features, consider, for example, rule 3.1 of algorithm 2: $S_3\ a\ n_j \rightarrow_{\texttt{min}} S_4\ a\ (c_i) \downarrow_j\ |\iota_i$. This rule uses complex symbols n_j and c_i, where j and i are free variables, which, in principle, could match anything, but, in this case, they will be only required to match cell IDs. Briefly, this rule, promoted by ι_i, consumes one a and one n_j, produces another a and sends down a c_i, where i is the index of the current cell, to child j, if this child exists.

3 Asynchronous P Systems

In traditional P systems, a universal clock is assumed to control the application of all rules, i.e. traditional P systems work synchronously, in *lock-step*. Practically, such universal clock is unrealistic in many distributed computing applications, where there is no such global clock and the communication delay is unpredictable.

Thus, it is interesting to investigate P systems that work in the asynchronous mode.

We define asynchronous P systems as follows. The rule format of asynchronous P systems is the same as for synchronous P systems, i.e., $s\ x \rightarrow_\alpha s'\ x'\ (y)_{\beta_\gamma}\ |_z$. However, we focus on typical distributed systems, where communications take substantially longer than actual local computations, therefore we consider that the message delay is totally unpredictable. In such systems, we assume that rules are applied in zero time and each message arrives in its own time t, $t \in [0, 1]$. Synchronous P systems are a special case of asynchronous P systems, where $t = 1$, for all evolutions. The *runtime complexity* of an asynchronous system is the *supremum* over all possible executions. We typically assume that messages sent over the same arc arrive in FIFO order (queue), or, as a possible extension—all messages sent over the same arc eventually arrive, but in arbitrary order (multiset).

We illustrate these concepts by means of a basic algorithm, **Echo** [22], in two distributed scenarios: (1) synchronous and (2) asynchronous, with a different (and less expected) evolution. Essentially, the Echo algorithm starts from a source cell, which broadcasts forward messages. These forward messages transitively reach all cells and, at the end, are reflected back to the initial source. The forward phase establishes a *virtual spanning tree* and the return phase is supposed to follow up its branches. The tree is only virtual, because it does not involve any structural changes; instead, virtual child-parent links are established by way of pointer objects.

Scenario 1 in Figure 1 assumes that all messages arrive in one time unit, i.e. in the synchronous mode. The forward and return phases take the same time, i.e. D time units each, where D is diameter of the undirected graph, G. Scenario 2 in Figure 2 assumes that some messages travel much faster than others, which is bad, but possible in asynchronous mode: $t = \epsilon$, where $0 < \epsilon \ll 1$. In this case, the forward and return phases take very different times, D and $N - 1$ time units, respectively, where N is the number of nodes of the undirected graph, G. The P system rules of the Echo algorithm are presented in Section 5.3.

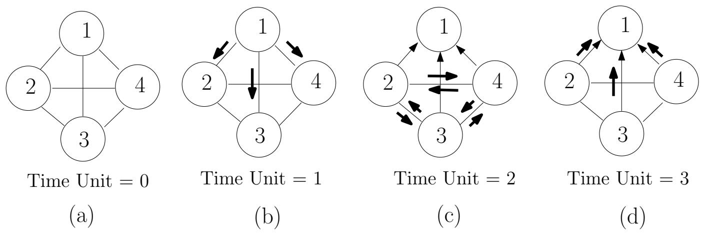

Fig. 1. Echo algorithm in synchronous mode—or in a “lucky” asynchronous mode, when all messages are propagated with the same delay (1). Arcs with arrows indicate child-parent arcs in the virtual spanning tree built by the algorithm. Thick arrows near arcs indicate messages.

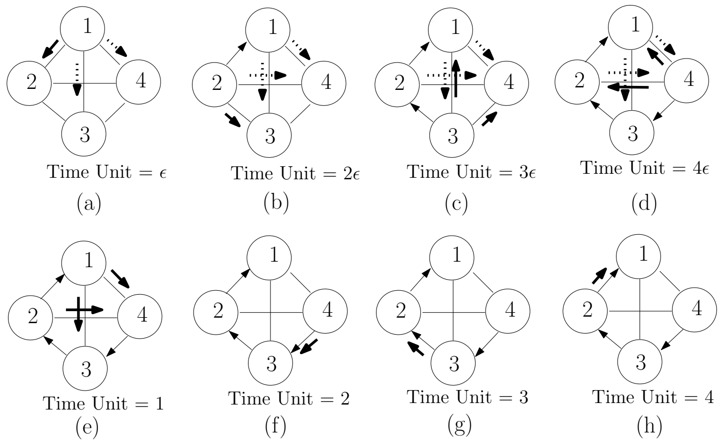

Fig. 2. Echo algorithm in asynchronous mode—one possible "bad" execution, among the many possible. Dotted thick arrows near arcs indicate messages still in transit.

4 Distributed Depth-First Search (DFS)

Depth-first search (DFS) and breadth-first search (BFS) are graph traversal algorithms, which construct a DFS spanning tree and a BFS spanning tree, respectively. Figure 3 shows the structure of a sample P system, Π, based on an "undirected" graph, G, and one possible *virtual* DFS spanning tree, T. We use quotation marks to indicate that G actually is a *directed* graph, but we do not care about arc orientation. The spanning tree is virtual, as it is described by "soft" pointer objects, not by "hard" structural arcs.

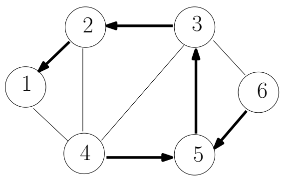

Fig. 3. P system Π based on an "undirected" graph and one possible virtual DFS spanning tree. Thick arrows indicate virtual child-parent arcs in this tree, linked by pointer objects.

DFS is a fundamental technique, inherently *sequential*, or so it appears. Several distributed DFS algorithms have been proposed, which attempt to make DFS run faster on distributed systems, such as the classical DFS [22], Awerbuch's DFS [1], Cidon's DFS [7], Sharma et al's DFS [21], Makki et al's DFS [15], Sense of Direction (SOD) DFS [22]. This is vast topic, which is impossible to present here at the required length. Therefore, we refer the reader to the original articles, or to a fundamental text, which covers all these algorithms, [22].

Several articles have proposed various synchronous P algorithms for DFS. Gutiérrez-Naranjo et al. proposed a DFS algorithm [12], using inhibitors to avoid visiting already-visited neighbor cells. Dinneen et al. [8] proposed a P algorithm to find disjoint paths in a digraph, using a distributed DFS strategy, which avoids visiting already-visited cells by changing the state of visited cells [9]. Bernardini et al. proposed a DFS algorithm in the P system synchronization problem [2]. This approach uses an operator, $mark_+$, to select one not-yet-visited cell, indicated by a 0 polarity, and then mark the cell as visited, by changing the polarity to +. In this case, the cell that performs a $mark_+$ operation, actually "knows" which child cell has been visited or not, without any message exchanges. In fact, all above mentioned P algorithms implement the classical DFS, which is discussed later in Section 4.2.

In the following sections, we present asynchronous P system implementations of the well-known distributed DFS algorithms, which leverage the parallel and distributed characteristics of P systems.

4.1 Discovering Neighbors

All our distributed DFS and BFS P algorithms, except the SoD algorithm, can, if needed, start with the same preliminary Phase I, in which cells discover their neighbors, i.e. their local topology. Nicolescu et al. have developed P algorithms to discover local topology and local neighbors [16, 9]. In this paper, we propose a crisper algorithm, Algorithm 1, with fewer symbols.

Algorithm 1 (Discovering cell neighbors)

Input: All cells start in the same initial state, S_0, with the same set of rules. Initially, each cell, σ_i, contains a cell ID object, ι_i, which is *immutable* and used as a *promoter*. Additionally, the source cell, σ_s, is decorated with one object a.

Output: All cells end in the same state, S_3. On completion, each cell contains the cell ID object, ι_i, and objects n_j, pointing to their neighbors. The source cell, σ_s, is still decorated with object a. Table 1 shows the neighborhoods of Figure 3, computed by Algorithm 1, in three P steps.

Table 1. Partial Trace of Algorithm 1 for Figure 3.

Step#	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	$S_0\ \iota_1 a$	$S_0\ \iota_2$	$S_0\ \iota_3$	$S_0\ \iota_4$	$S_0\ \iota_5$	$S_0\ \iota_6$
3	$S_3\ \iota_1 a n_2 n_4$	$S_3\ \iota_2 n_1 n_3 n_4$	$S_3\ \iota_3 n_2 n_4 n_5 n_6$	$S_3\ \iota_4 n_1 n_2 n_3 n_5$	$S_3\ \iota_5 n_3 n_4 n_6$	$S_3\ \iota_6 n_3 n_5$

0. Rules in state S_0:
   1 $S_0\ a \rightarrow_{\texttt{min}} S_1\ ay\ (z) \updownarrow$
   2 $S_0\ z \rightarrow_{\texttt{min}} S_1\ y\ (z) \updownarrow$
   3 $S_0\ z \rightarrow_{\texttt{max}} S_1$
1. Rules in state S_1:
   1 $S_1\ y \rightarrow_{\texttt{min}} S_2\ (n_i) \updownarrow |_{\iota_i}$
   2 $S_1\ z \rightarrow_{\texttt{max}} S_2$
2. Rules for state S_2:
   1 $S_2 \rightarrow_{\texttt{min}} S_3$
   2 $S_2\ z \rightarrow_{\texttt{max}} S_3$

In state S_0, the source cell, σ_s, which is decorated by object a, broadcasts signal z, to all cells, and enters state S_1. Each cell receiving z produces one object y, and changes to state S_1. Superfluous signals z are discarded. Then, in state S_1, each cell that has object y, sends its own ID, which appears as subscript in complex object n_i, to all its neighbors. In state S_2, cells accumulate the received neighbor objects, discard superfluous objects z, and enter S_3.

4.2 Classical DFS

The classical DFS algorithm is based on Tarry's traversal algorithm, which traverses all arcs *sequentially*, in both directions, using a visiting *token* [22]. Because it traverses all arcs twice, serially, the classical DFS algorithm is not the most efficient distributed DFS algorithm.

Algorithm 2 (Classical DFS)

Input: All cells start in the same quiescent state, S_3, and with the same set of rules. Each cell, σ_i, contains an immutable cell ID object, ι_i. All cells know their neighbors, i.e. they have topological awareness, which are indicated by pointer objects, n_j (as built by Algorithm 1). The source cell, σ_s, is additionally decorated with one object, a, which triggers the search.

Output: All cells end in the same final state (S_5). On completion, the cell IDs are intact. Cell σ_s is still decorated with one a and all other cells contain DFS spanning tree pointer objects, indicating predecessors, p_j.

Table 2 shows one possible DFS spanning tree, built by this algorithm, for the P system Π of Figure 3.

Table 2. Partial Trace of Algorithm 2 for Figure 3.

Step#	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	$S_3\ \iota_1 a n_2 n_4$	$S_3\ \iota_2 n_1 n_3 n_4$	$S_3\ \iota_3 n_2 n_4 n_5 n_6$	$S_3\ \iota_4 n_1 n_2 n_3 n_5$	$S_3\ \iota_5 n_3 n_4 n_6$	$S_3\ \iota_6 n_3 n_5$
19	$S_5\ \iota_1 a$	$S_5\ \iota_2 p_1$	$S_5\ \iota_3 p_2$	$S_5\ \iota_4 p_5$	$S_5\ \iota_5 p_3$	$S_5\ \iota_6 p_5$

3. Rules in state S_3:
   1 $S_3\ an_j \rightarrow_{\texttt{min}} S_4\ a\ (c_i) \downarrow_j |_{\iota_i}$
   2 $S_3\ c_j n_j n_k \rightarrow_{\texttt{min}} S_4\ p_j\ (c_i) \downarrow_k |_{\iota_i}$
4. Rules for state S_4:
   1 $S_4\ c_j n_j \rightarrow_{\texttt{min}} S_4\ (x_i) \downarrow_j |_{\iota_i}$
   2 $S_4\ x_j n_k \rightarrow_{\texttt{min}} S_4\ (c_i) \downarrow_k |_{\iota_i}$
   3 $S_4\ x_j p_k \rightarrow_{\texttt{min}} S_5\ p_k\ (x_i) \downarrow_k |_{\iota_i}$
   4 $S_4\ x_j \rightarrow_{\texttt{min}} S_5$

4.3 Awerbuch DFS

Awerbuch's algorithm [1] and other more efficient algorithms improve time complexity by having the visiting token traversing tree arcs only, all other arcs are traversed in parallel, by auxiliary messages. Specifically, in Awerbuch's algorithm, when the node is visited for the first time, it *notifies* all neighbors that it has been visited and waits until it receives all neighbors' *acknowledgments*. After that, the node can visit one of its unvisited neighbors. Thus, the node knows exactly which of its neighbors have been visited and avoids visiting the already-visited neighbors, which saves time.

Algorithm 3 (Awerbuch DFS)

Input: Same as in Algorithm 2.

Output: Similar to Algorithm 2, but the final state is S_7. Also, cells may contain "garbage" objects, which can be cleared, by using a few more steps.

Table 3 shows the resulting DFS spanning tree, for Figure 3. Table 16 from Appendix A contains full traces for this algorithm, including the preliminary phase, of Algorithm 1.

Table 3. Partial Trace of Algorithm 3 for Figure 3.

Step#	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	$S_3\ \iota_1 a n_2 n_4$	$S_3\ \iota_2 n_1 n_3 n_4$	$S_3\ \iota_3 n_2 n_4 n_5 n_6$	$S_3\ \iota_4 n_1 n_2 n_3 n_5$	$S_3\ \iota_5 n_3 n_4 n_6$	$S_3\ \iota_6 n_3 n_5$
24	$S_7\ \iota_1 a\ \ldots$	$S_7\ \iota_2 p_1\ \ldots$	$S_7\ \iota_3 p_2\ \ldots$	$S_7\ \iota_4 p_5\ \ldots$	$S_7\ \iota_5 p_3\ \ldots$	$S_7\ \iota_6 p_5\ \ldots$

3. Rules in state S_3:
   1 $S_3\ n_j \rightarrow_{\texttt{min}} S_4\ n_j m_j$
4. Rules in state S_4:
   1 $S_4\ v_j \rightarrow_{\texttt{min}} S_4\ u_j\ (b_i) \downarrow_j |_{\iota_i}$
   2 $S_4\ n_j \rightarrow_{\texttt{min}} S_5\ n_j\ (v_i) \downarrow_j |_{a\iota_i}$
   3 $S_4\ c_j m_j n_j \rightarrow_{\texttt{min}} S_5\ p_j$
   4 $S_4\ n_j \rightarrow_{\texttt{min}} S_5\ n_j\ (v_i) \downarrow_j |_{t\iota_i}$
5. Rules for state S_5:
   1 $S_5\ n_j \rightarrow_{\texttt{min}} S_6\ n_j w_j$
6. Rules for state S_6:
   1 $S_6\ w_j \rightarrow_{\texttt{min}} S_7 |_{b_j}$
   2 $S_6\ w_j p_k \rightarrow_{\texttt{min}} S_7\ w_j p_k |_{b_l}$
   3 $S_6\ b_j \rightarrow_{\texttt{min}} S_7$
   4 $S_6\ u_j m_j \rightarrow_{\texttt{min}} S_7\ u_j$
   5 $S_6\ am_j \rightarrow_{\texttt{min}} S_7\ au_j\ (c_i t) \downarrow_j |_{\iota_i}$
   6 $S_6\ p_k m_j \rightarrow_{\texttt{min}} S_7\ p_k u_j\ (c_i t) \downarrow_j |_{\iota_i}$
   7 $S_6\ p_j \rightarrow_{\texttt{min}} S_7\ p_j\ (x_i t) \downarrow_j |_{\iota_i}$
   8 $S_6\ t \rightarrow_{\texttt{min}} S_7$
7. Rules for state S_7:

1 $S_7\ w_j \rightarrow_{\texttt{min}} S_7\ |_{b_j}$
2 $S_7\ w_j p_k \rightarrow_{\texttt{min}} S_7\ w_j p_k$
3 $S_7\ p_k m_j \rightarrow_{\texttt{min}}$
$S_7\ p_k u_j\ (c_i t) \downarrow_j |_{b_l \iota_i}$
4 $S_7\ p_j \rightarrow_{\texttt{min}} S_7\ p_j\ (x_i t) \downarrow_j |_{b_l \iota_i}$
5 $S_7\ b_j \rightarrow_{\texttt{min}} S_7$
6 $S_7\ m_k x_j \rightarrow_{\texttt{min}} S_7\ u_k\ (c_i t) \downarrow_k |_{\iota_i}$
7 $S_7\ p_k x_j \rightarrow_{\texttt{min}} S_7\ p_k\ (x_i t) \downarrow_k |_{\iota_i}$
8 $S_7\ v_j \rightarrow_{\texttt{min}} S_7\ u_j\ (b_i) \downarrow_j |_{\iota_i}$
9 $S_7\ u_j m_j \rightarrow_{\texttt{min}} S_7\ u_j$
10 $S_7\ a x_j \rightarrow_{\texttt{min}} S_7\ a$
11 $S_7\ t \rightarrow_{\texttt{min}} S_7$

4.4 Cidon DFS

Cidon's algorithm [7] improves Awerbuch's algorithm by not using *acknowledgments*, therefore removing a delay. The token holding cell does not wait for the neighbors' acknowledgments, but immediately visits a neighbor. However, it needs to record the most recent neighbor used, to solve cases when visiting *notifications* arrive after the visiting *token*.

Algorithm 4 (Cidon DFS)

Input: Same as in Algorithm 2.

Output: Similar to Algorithm 2, but the final state is S_5. Also, cells may contain "garbage" objects, which can be cleared, by using a few more steps.

Table 4 shows one possible DFS spanning tree, built by this algorithm, for the P system Π of Figure 3.

Table 4. Partial Trace of Algorithm 4 for Figure 3.

Step#	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	$S_3\ \iota_1 a n_2 n_4$	$S_3\ \iota_2 n_1 n_3 n_4$	$S_3\ \iota_3 n_2 n_4 n_5 n_6$	$S_3\ \iota_4 n_1 n_2 n_3 n_5$	$S_3\ \iota_5 n_3 n_4 n_6$	$S_3\ \iota_6 n_3 n_5$
12	$S_5\ \iota_1 a\ \ldots$	$S_5\ \iota_2 p_1\ \ldots$	$S_5\ \iota_3 p_2\ \ldots$	$S_5\ \iota_4 p_5\ \ldots$	$S_5\ \iota_5 p_3\ \ldots$	$S_5\ \iota_6 p_5\ \ldots$

3. Rules in state S_3:
   1 $S_3\ n_j \rightarrow_{\texttt{min}} S_4\ n_j m_j$
   2 $S_3\ a \rightarrow_{\texttt{min}} S_4\ at$
4. Rules in state S_4:
   1 $S_4\ a n_j m_j \rightarrow_{\texttt{min}}$
   $S_5\ a v_j\ (v_i c_i t) \downarrow_j |_{t \iota_i}$
   2 $S_4\ c_k n_k m_k n_j m_j \rightarrow_{\texttt{min}}$
   $S_5\ p_k r_j m_j\ (v_i c_i t) \downarrow_j |_{t \iota_i}$
   3 $S_4\ c_k m_k n_j m_j \rightarrow_{\texttt{min}}$
   $S_5\ p_k r_j m_j\ (v_i c_i t) \downarrow_j |_{t \iota_i}$
   4 $S_4\ c_j n_j m_j \rightarrow_{\texttt{min}} S_5\ p_j\ (x_i t) \downarrow_j |_{t \iota_i}$
   5 $S_4\ c_j m_j \rightarrow_{\texttt{min}} S_5\ p_j\ (x_i t) \downarrow_j |_{t \iota_i}$
   6 $S_4\ m_j \rightarrow_{\texttt{min}} S_5\ m_j\ (v_i) \downarrow_j |_{t \iota_i}$
   7 $S_4\ v_j n_j \rightarrow_{\texttt{min}} S_4\ v_j$
   8 $S_4\ t \rightarrow_{\texttt{min}} S_5$
5. Rules for state S_5:
   1 $S_5\ r_k v_k n_j \rightarrow_{\texttt{min}} S_5\ r_j\ (c_i t) \downarrow_j |_{\iota_i}$
   2 $S_5\ r_k v_k p_j \rightarrow_{\texttt{min}} S_5\ p_j\ (x_i t) \downarrow_j |_{\iota_i}$
   3 $S_5\ x_j n_k m_k \rightarrow_{\texttt{min}}$
   $S_5\ r_k m_k\ (v_i c_i t) \downarrow_k |_{t \iota_i}$
   4 $S_5\ x_j p_k r_j \rightarrow_{\texttt{min}}$
   $S_5\ p_k r_j\ (x_i t) \downarrow_k |_{t \iota_i}$
   5 $S_5\ c_j p_k \rightarrow_{\texttt{min}} S_5\ p_k v_j$
   6 $S_5\ v_j n_j \rightarrow_{\texttt{min}} S_5\ v_j$
   7 $S_5\ a x_j \rightarrow_{\texttt{min}} S_5\ a$
   8 $S_5\ t \rightarrow_{\texttt{min}} S_5$

4.5 Sharma DFS

Sharma et al.'s algorithm [21] further improves time complexity, at the cost of increasing the message size, by including a *list* of visited nodes when passing the visiting *token* [23]. Thus, it eliminates unnecessary message exchanges to inform neighbors of visited status.

Algorithm 5 (Sharma DFS)

Input: Same as in Algorithm 2.

Output: Similar to Algorithm 2, but the final state is S_4. Also, cells may contain "garbage" objects, which can be cleared, by using a few more steps.

Table 5 shows one possible DFS spanning tree, built by this algorithm, for the P system Π of Figure 3.

Table 5. Partial Trace of Algorithm 5 for Figure 3.

Step#	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	$S_3\ \iota_1 a n_2 n_4$	$S_3\ \iota_2 n_1 n_3 n_4$	$S_3\ \iota_3 n_2 n_4 n_5 n_6$	$S_3\ \iota_4 n_1 n_2 n_3 n_5$	$S_3\ \iota_5 n_3 n_4 n_6$	$S_3\ \iota_6 n_3 n_5$
11	$S_4\ \iota_1 a\ \ldots$	$S_4\ \iota_2 p_1\ \ldots$	$S_4\ \iota_3 p_2\ \ldots$	$S_4\ \iota_4 p_5\ \ldots$	$S_4\ \iota_5 p_3\ \ldots$	$S_4\ \iota_6 p_5\ \ldots$

3. Rules in state S_3:
   1 $S_3\ a n_j \rightarrow_{\texttt{min}} S_4\ a\ (c_i v_i t) \downarrow_j |_{\iota_i}$
   2 $S_3\ n_j \rightarrow_{\texttt{min}} S_4\ |_{t v_j}$
   3 $S_3\ c_j \rightarrow_{\texttt{min}} S_4\ p_j\ (c_i v_i t) \downarrow_k |_{n_k \iota_i}$
   4 $S_3\ c_j \rightarrow_{\texttt{min}} S_4\ p_j\ (x_i v_i v_j t) \downarrow_j |_{\iota_i}$
   5 $S_3\ v_j \rightarrow_{\texttt{min}} S_4\ v_j\ (v_j) \downarrow_k |_{t n_k}$
   6 $S_3\ t \rightarrow_{\texttt{min}} S_4$
4. Rules for state S_4:
   1 $S_4\ n_j \rightarrow_{\texttt{min}} S_4\ v_j$
   2 $S_4\ x_j \rightarrow_{\texttt{min}} S_4\ (c_i v_i t) \downarrow_k |_{n_k \iota_i}$
   3 $S_4\ x_j \rightarrow_{\texttt{min}} S_4\ (x_i v_i t) \downarrow_k |_{p_k \iota_i}$
   4 $S_4\ v_j \rightarrow_{\texttt{min}} S_4\ v_j\ (v_j) \downarrow_k |_{t n_k}$
   5 $S_4\ v_j \rightarrow_{\texttt{min}} S_4\ v_j\ (v_j) \downarrow_k |_{t p_k}$
   6 $S_4\ t \rightarrow_{\texttt{min}} S_4$
   7 $S_4\ a x_j \rightarrow_{\texttt{min}} S_4\ a$

4.6 Makki DFS

Makki et al.'s algorithm [15] improves Sharma et al.'s algorithm by using a *dynamic backtracking* technique. It keeps track of the most recent *split point*, i.e. the lowest ancestor node. When the search path backtracks to a node, if the node has a non-tree edge to its split point, it backtracks to the split point directly via that edge, rather than following the longer tree path to its split point.

Algorithm 6 (Makki DFS)

Input: Same as in Algorithm 2.

Output: Similar to Algorithm 2, but the final state is S_4. Also, cells may contain "garbage" objects, which can be cleared, by using a few more steps.

Table 6 shows one possible DFS spanning tree, built by this algorithm, for the P system Π of Figure 3.

Table 6. Partial Trace of Algorithm 6 for Figure 3.

Step#	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	$S_3\ \iota_1 a n_2 n_4$	$S_3\ \iota_2 n_1 n_3 n_4$	$S_3\ \iota_3 n_2 n_4 n_5 n_6$	$S_3\ \iota_4 n_1 n_2 n_3 n_5$	$S_3\ \iota_5 n_3 n_4 n_6$	$S_3\ \iota_6 n_3 n_5$
10	$S_4\ \iota_1 a\ \ldots$	$S_4\ \iota_2 p_1\ \ldots$	$S_4\ \iota_3 p_2\ \ldots$	$S_4\ \iota_4 p_5\ \ldots$	$S_4\ \iota_5 p_3\ \ldots$	$S_4\ \iota_6 p_5\ \ldots$

3. Rules in state S_3:
   1 $S_3\ an_j \rightarrow_{\texttt{min}} S_4\ a\ (c_i v_i s_i t) \downarrow_j |_{\iota_i}$
   2 $S_3\ n_j \rightarrow_{\texttt{min}} S_4\ |_{tv_j}$
   3 $S_3\ c_j s_m \rightarrow_{\texttt{min}}$
   $S_4\ p_j r_m\ (c_i v_i s_i t) \downarrow_k |_{n_k n_l \iota_i}$
   4 $S_3\ c_j s_l \rightarrow_{\texttt{min}}$
   $S_4\ p_j r_l\ (c_i v_i s_l t) \downarrow_k |_{n_k \iota_i}$
   5 $S_3\ c_j \rightarrow_{\texttt{min}} S_4\ p_j r_k\ (x_i v_i t) \downarrow_k |_{s_k \iota_i}$
   6 $S_3\ c_j \rightarrow_{\texttt{min}} S_4\ p_j r_k\ (x_i v_i t) \downarrow_j |_{s_k \iota_i}$
   7 $S_3\ v_j \rightarrow_{\texttt{min}} S_4\ v_j\ (v_j) \downarrow_k |_{tn_k}$
   8 $S_3\ v_j \rightarrow_{\texttt{min}} S_4\ v_j\ (v_j) \downarrow_k |_{ts_k}$
   9 $S_3\ t \rightarrow_{\texttt{min}} S_4$
4. Rules for state S_4:
   1 $S_4\ n_j \rightarrow_{\texttt{min}} S_4\ v_j$
   2 $S_4\ x_j \rightarrow_{\texttt{min}} S_4\ (c_i v_i s_i t) \downarrow_k |_{n_k n_l \iota_i}$
   3 $S_4\ x_j r_l \rightarrow_{\texttt{min}}$
   $S_4\ (c_i v_i s_i s_l t) \downarrow_k |_{n_k \iota_i}$
   4 $S_4\ x_j \rightarrow_{\texttt{min}} S_4\ (x_i v_i t) \downarrow_k |_{r_k \iota_i}$
   5 $S_4\ x_j \rightarrow_{\texttt{min}} S_4\ (x_i v_i t) \downarrow_k |_{p_k \iota_i}$
   6 $S_4\ v_j \rightarrow_{\texttt{min}} S_4\ v_j\ (v_j) \downarrow_k |_{tn_k}$
   7 $S_4\ v_j \rightarrow_{\texttt{min}} S_4\ v_j\ (v_j) \downarrow_k |_{tr_k}$
   8 $S_4\ v_j \rightarrow_{\texttt{min}} S_4\ v_j\ (v_j) \downarrow_k |_{tp_k}$
   9 $S_4\ t \rightarrow_{\texttt{min}} S_4$
   10 $S_4\ ax_j \rightarrow_{\texttt{min}} S_4\ a$

4.7 Sense of Direction DFS

With Sense of Direction (SOD), the node labeling is not required. Instead, arc labeling is used, with the following properties:

- Edges are labeled with elements of a group G, typically $G = Z_n$, where $Z_n = \{0, 1, \ldots, n-1\}$.
- Given labeled arcs $a_0 \overset{x_1}{\rightarrow} a_1, a_1 \overset{x_2}{\rightarrow} a_2, \ldots a_{k-1} \overset{x_k}{\rightarrow} a_k$, the path $a_0 \overset{x_1}{\rightarrow} a_1 \overset{x_2}{\rightarrow} a_2 \ldots a_{k-1} \overset{x_k}{\rightarrow} a_k$ has label $x_1 + x_2 + \ldots + x_k$.
- Given labelled paths $a \overset{x}{\Rightarrow} b$ and $c \overset{x}{\Rightarrow} d$, $a = c$ if and only if $b = d$.

Thus, in search algorithms, path labels can very handily indicate the already-visited nodes. Path labels are kept as a growing list and are appended when the search path passes a node.

If the search path reaching the node, a_k, wants to visit the node, a_{k+1}, it first checks whether a_{k+1} is an already-visited node, e.g., a_i, $0 \leq i \leq n$. The node a_k checks whether one of the partial path labels, e.g., $x_{i+1} + \ldots + x_k + x_{k+1}$, equals zero. If yes, then $a_{k+1} = a_i$, thus a_{k+1} is an already-visited node. We refer the readers to [22] for more details about SOD.

Figure 4 shows a sample P system based on directed graph with SOD arc labels.

Algorithm 7 (Sense of Direction DFS)

For this particular algorithm, here, we only present a P system-like high-level pseudo-code. Additional investigation is required to achieve an efficient translation to usual rewriting rules.

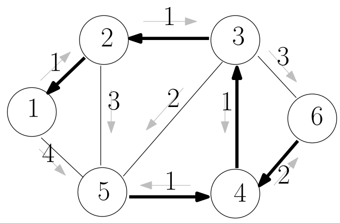

Fig. 4. A sample P system based on a SOD structure, with arc labelling, indicated by gray arrows. Thick arc arrows indicate a possible virtual DFS tree.

Input: All cells start with the same set of rules and start in the same quiescent state, S_0. Initially, all cells contain objects indicating the labels of neighbor arcs: objects o_j for outgoing arcs and objects e_j for incoming arcs. The source cell, σ_s, is additionally decorated with one trigger object, a.

Output: All cells end in the same final state, S_1. On completion, cell σ_s is still decorated with one a. All other cells contain DFS spanning tree pointer objects, indicating its tree predecessors: p_j, for incoming arcs and q_j, for outgoing arcs. Also, cells may contain "garbage" objects, which can be cleared, in a few more steps.

Table 7 shows one possible DFS spanning tree, built by this algorithm, for the P system of Figure 4.

Table 7. Partial Trace of Algorithm 7 for Figure 4.

Step#	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	$S_0 a o_1 o_4$	$S_0 e_1 o_1 o_3$	$S_0 e_1 o_1 o_2 o_3$	$S_0 e_1 o_1 o_2$	$S_0 e_1 e_2 e_3 e_4$	$S_0 e_2 e_3$
11	$S_1\ a\ \ldots$	$S_1\ p_1\ \ldots$	$S_1\ p_1\ \ldots$	$S_1\ p_1\ \ldots$	$S_1\ p_1\ \ldots$	$S_1\ p_2\ \ldots$

The ruleset below uses a few additional "magical" algebraic operators and prompters, which do fit properly into the basic framework outlined in Section 2 (or not yet).

- Operation $\pi \oplus j$ adds j, modulo n, to every element in list π and also appends $+j$ to list π.
- Operation $\pi \ominus j$ subtracts j, modulo n, from every element in list π and also appends $n - j$ (i.e. $-j$ modulo n) to list π.
- Complex promoters $\pi \oplus j?$ and $\pi \ominus j?$ enable the associated rule only if the resulting list does not contain any 0.

0. Rules in state S_0:
   1 $S_0\ a o_j \rightarrow_{\texttt{min}} S_1\ a\ (c_j b_{\oplus j}) \uparrow_j$
   2 $S_0\ b_\pi o_j c_k e_k \rightarrow_{\texttt{min}}$
   $S_1\ p_k (c_j b_{\pi \oplus j}) \uparrow_j |_{\pi \oplus j?}$
   3 $S_0\ b_\pi e_j c_k e_k \rightarrow_{\texttt{min}}$
   $S_1\ p_k (l_j b_{\pi \ominus j}) \downarrow_j |_{\pi \ominus j?}$
   4 $S_0\ b_\pi o_j l_k o_k \rightarrow_{\texttt{min}}$
   $S_1\ q_k (c_j b_{\pi \oplus j}) \uparrow_j |_{\pi \oplus j?}$

5 $S_0\ b_\pi e_j l_k o_k \rightarrow_{\mathtt{min}}$
$S_1\ q_k(l_j b_{\pi \ominus j}) \downarrow_j |_{\pi \ominus j?}$
6 $S_0\ b_\pi c_j e_j \rightarrow_{\mathtt{min}} S_1\ p_j(x_j b_{\pi \ominus j}) \downarrow_j$
7 $S_0\ b_\pi l_j o_j \rightarrow_{\mathtt{min}} S_1\ p_j(x_j b_{\pi \oplus j}) \uparrow_j$

1. Rules in state S_1:

1 $S_1\ b_\pi x_k o_j \rightarrow_{\mathtt{min}}$
$S_1\ (c_j b_{\pi \oplus j}) \uparrow_j |_{\pi \oplus j?}$
2 $S_1\ b_\pi x_k e_j \rightarrow_{\mathtt{min}}$
$S_1\ (l_j b_{\pi \ominus j}) \downarrow_j |_{\pi \ominus j?}$
3 $S_1\ b_\pi x_k p_j \rightarrow_{\mathtt{min}} S_1\ p_j\ (x_j b_{\pi \ominus j}) \downarrow_j$
4 $S_1\ b_\pi x_k q_j \rightarrow_{\mathtt{min}} S_1\ q_j\ (x_j b_{\pi \oplus j}) \uparrow_j$
5 $S_1\ a x_j \rightarrow_{\mathtt{min}} S_1\ a$

5 Distributed Breadth-First Search (BFS)

BFS is a fundamental technique, inherently *parallel*, or so it appears. There are a number of distributed BFS algorithms to make BFS run faster on parallel and distributed systems, such as Synchronous BFS [22], Asynchronous BFS [22], an improved Asynchronous BFS with known graph diameter [22], Layered BFS [22], Hybrid BFS [22].

Our previous research proposed a P algorithm to find disjoint paths using BFS, and empirical results show that BFS can leverage the parallel and distributed characteristics of P systems [17]. In this paper, we first present a P implementation of synchronous BFS (SyncBFS) and discuss how SyncBFS succeeds in the synchronous mode but *fails* in the asynchronous mode. Next, we propose a P implementation of an algorithm which works correctly in the asynchronous mode, the simple Asynchronous BFS (AsyncBFS) algorithm, and we show how it works in both synchronous and asynchronous scenarios.

5.1 Synchronous BFS

Initially, the source cell broadcasts out a search token. On receiving the search token, an unmarked cell marks itself and chooses one of the cells from which the search token arrived as its parent. Then in the first round after the cell gets marked, it broadcasts a search token to all its neighbors [14]. SyncBFS is a "wave" algorithm and it produces a BFS spanning tree in synchronous mode, as shown in Figure 5. However, it often fails in asynchronous mode, as shown in Figure 6.

Algorithm 8 (Synchronous BFS)

Input: Same as in Algorithm 2.

Synchronous output: All cells end in the same final state, S_5. On completion, each cell, σ_i, still contains its cell ID object, ι_i. The source cell, σ_s, is still decorated with one a. All other cells contain BFS spanning tree pointer objects, indicating predecessors, p_j. Also, cells may contain "garbage" objects, which can be cleared, by using a few more steps.

Table 8 shows the BFS spanning tree built by this algorithm (in the synchronous mode), for the P system of Figure 5 (there is only one BFS tree in this case).

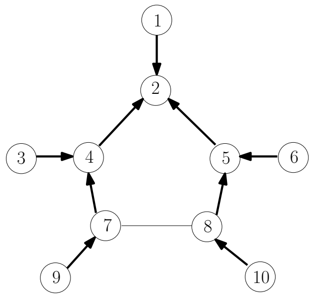

Fig. 5. BFS spanning tree.

Table 8. Partial Trace of Algorithm 8 for Figure 5 in synchronous mode.

Step#	σ_1	σ_2	σ_3	σ_4	σ_5
0	$S_3\ \iota_1 n_2$	$S_3\ \iota_2 n_1 n_4 n_5$	$S_3\ \iota_3 n_4$	$S_3\ \iota_4 n_2 n_3 n_7$	$S_3\ \iota_5 n_2 n_6 n_8$
8	$S_5\ \iota_1 p_2 \ldots$	$S_5\ \iota_2 a \ldots$	$S_5\ \iota_3 p_4 \ldots$	$S_5\ \iota_4 p_2 \ldots$	$S_5\ \iota_5 p_2 \ldots$

Step#	σ_6	σ_7	σ_8	σ_9	σ_{10}
0	$S_3\ \iota_6 n_5$	$S_3\ \iota_7 n_4 n_8 n_9$	$S_3\ \iota_8 n_5 n_7 n_{10}$	$S_3\ \iota_9 n_7$	$S_3\ \iota_{10} n_8$
8	$S_5\ \iota_6 p_5 \ldots$	$S_5\ \iota_7 p_4 \ldots$	$S_5\ \iota_8 p_5 \ldots$	$S_5\ \iota_9 p_7 \ldots$	$S_5\ \iota_{10} p_8 \ldots$

3. Rules in state S_3:
   1 $S_3\ a \rightarrow_{\mathtt{min}} S_4\ a$
   2 $S_3\ c_j n_j \rightarrow_{\mathtt{min}} S_4\ p_j$
4. Rules for state S_4:
   1 $S_4\ n_j \rightarrow_{\mathtt{min}} S_5\ (c_i) \downarrow_j |_{\iota_i}$
   2 $S_4 \rightarrow_{\mathtt{min}} S_5$
5. Rules for state S_5:
   1 $S_5\ c_j \rightarrow_{\mathtt{min}} S_5$

However, if Algorithm 8 runs in asynchronous mode, the result is still a *spanning tree*, but *not* necessarily a *BFS spanning tree*, as illustrated in Table 9 and Figure 6. The search token from cell σ_2 to σ_5 is delayed and arrives in cell σ_5 after σ_5 records its parent as σ_8. The resulting spanning tree is not a BFS spanning tree.

Table 9. Partial Trace of Algorithm 8 for Figure 6 in asynchronous mode.

Step#	σ_1	σ_2	σ_3	σ_4	σ_5
0	$S_3\ \iota_1 n_2$	$S_3\ \iota_2 n_1 n_4 n_5$	$S_3\ \iota_3 n_4$	$S_3\ \iota_4 n_2 n_3 n_7$	$S_3\ \iota_5 n_2 n_6 n_8$
14	$S_5\ \iota_1 p_1 \ldots$	$S_5\ \iota_2 a \ldots$	$S_5\ \iota_3 p_4 \ldots$	$S_5\ \iota_4 p_2 \ldots$	$S_5\ \iota_5 p_8 \ldots$

Step#	σ_6	σ_7	σ_8	σ_9	σ_{10}
0	$S_3\ \iota_6 n_5$	$S_3\ \iota_7 n_4 n_8 n_9$	$S_3\ \iota_8 n_5 n_7 n_{10}$	$S_3\ \iota_9 n_7$	$S_3\ \iota_{10} n_8$
14	$S_5\ \iota_6 p_5 \ldots$	$S_5\ \iota_7 p_4 \ldots$	$S_5\ \iota_8 p_7 \ldots$	$S_5\ \iota_9 p_7 \ldots$	$S_5\ \iota_{10} p_8 \ldots$

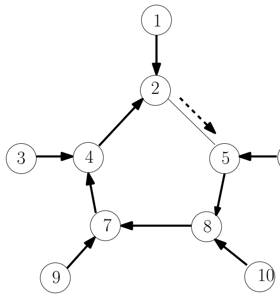

Fig. 6. BFS spanning tree output of Algorithm 8 in an asynchronous scenario.

5.2 Asynchronous BFS

Asynchronous BFS (AsyncBFS) algorithm is not just a asynchronous version of SyncBFS [14], as previously discussed in the asynchronous mode of SyncBFS. It has modifications to correct the parent destination, therefore obtaining a BFS spanning tree.

Although the known problem of AsyncBFS is that there is no way to know when there are no further parent corrections to make, i.e. it never produces the tree structure output. However, in P systems, there is no such problem, because the objects in cells are actually the tree link output. Thus, P systems provides a favorable way to implement this algorithm, which does not require other augmenting approaches, such as adding acknowledgments, convergecasting acknowledgments, bookkeeping, etc [14].

Algorithm 9 (Asynchronous BFS)

Input: Same as in Algorithms 2 (and 8).

Output: Similar to Algorithm 8 (running in synchronous mode), but the final state is S_4. Also, cells may contain "garbage" objects, which can be cleared, by using a few more steps.

Table 10 shows the BFS spanning tree built by this algorithm, for the P system of Figure 5 (there is only one BFS tree in this case).

Table 10. Partial Trace of Algorithm 9 for Figure 5.

Step#	σ_1	σ_2	σ_3	σ_4	σ_5
0	$S_3\ \iota_1 n_2$	$S_3\ \iota_2 n_1 n_4 n_5$	$S_3\ \iota_3 n_4$	$S_3\ \iota_4 n_2 n_3 n_7$	$S_3\ \iota_5 n_2 n_6 n_8$
5	$S_4 p_2 \ldots$	$S_4 a \ldots$	$S_4 p_4 \ldots$	$S_4 p_2 \ldots$	$S_4 p_2 \ldots$
Step#	σ_6	σ_7	σ_8	σ_9	σ_{10}
0	$S_3\ \iota_6 n_5$	$S_3\ \iota_7 n_4 n_8 n_9$	$S_3\ \iota_8 n_5 n_7 n_{10}$	$S_3\ \iota_9 n_7$	$S_3\ \iota_{10} n_8$
5	$S_4\ \iota_6 p_5 \ldots$	$S_4\ \iota_7 p_4 \ldots$	$S_4\ \iota_8 p_5 \ldots$	$S_4\ \iota_9 p_7 \ldots$	$S_4\ \iota_{10} p_8 \ldots$

3. Rules in state S_3:
   1 $S_3 \ \rightarrow_{\texttt{min}} S_4 \ h|_a$
   2 $S_3 \ n_j \rightarrow_{\texttt{min}}$
   $S_4 \ m_j \ (c_i tgguu) \downarrow_j |_{a\iota_i}$
   3 $S_3 \ c_j n_j \rightarrow_{\texttt{min}} S_4 \ p_j m_j|_t$
   4 $S_3 \ \rightarrow_{\texttt{min}} S_4 \ (c_i t) \downarrow_j |_{tn_j\iota_i}$
   5 $S_3 \ gu \rightarrow_{\texttt{min}} S_4 \ h \ (gguu) \updownarrow |_t$
   6 $S_3 \ gu \rightarrow_{\texttt{max}} S_4 \ h \ (gu) \updownarrow |_t$
   7 $S_3 \ n_j \rightarrow_{\texttt{min}} S_4 \ m_j|_t$
   8 $S_3 \ t \rightarrow_{\texttt{max}} S_4$
4. Rules for state S_4:
   1 $S_4 \ gh \rightarrow_{\texttt{max}} S_4 \ |_t$
   2 $S_4 \ p_j \rightarrow_{\texttt{min}} S_4 \ |_{ht}$
   3 $S_4 \ c_j m_j \rightarrow_{\texttt{min}} S_4 \ p_j|_{ht}$
   4 $S_4 \ c_j \rightarrow_{\texttt{min}} S_4 \ |_t$
   5 $S_4 \ m_j \rightarrow_{\texttt{min}} S_4 \ (c_i t) \downarrow_j |_{ht\iota_i}$
   6 $S_4 \ u \rightarrow_{\texttt{min}} S_4 \ h \ (gguu) \updownarrow |_{ht}$
   7 $S_4 \ u \rightarrow_{\texttt{max}} S_4 \ h \ (gu) \updownarrow |_{ht}$
   8 $S_4 \ h \rightarrow_{\texttt{max}} S_4 \ |_t$
   9 $S_4 \ gu \rightarrow_{\texttt{max}} S_4$
   10 $S_4 \ gu \rightarrow_{\texttt{max}} S_4|_t$
   11 $S_4 \ t \rightarrow_{\texttt{max}} S_4$

5.3 Echo Algorithm

The Echo algorithm shares the similar "wave" characteristics of distributed BFS algorithms, but, as discussed in Section 3, it only builds a spanning tree, not necessarily a BFS spanning tree.

Algorithm 10 (Echo Algorithm)

Input: Same as in Algorithms 2 (and 8).

Output: All cells end in the same final state, S_4. On completion, each cell, σ_i, still contains its cell ID object, ι_i. he source cell, σ_s, is still decorated with an object, a. All other cells contain a spanning tree pointer objects, indicating predecessors, p_j.

Table 11 and 12 show two spanning trees, built by this algorithm, for the P system of Figures 1 and 2, in synchronous and asynchronous modes, respectively.

Table 11. Partial Trace of Algorithm 10 for Figure 1 in synchronous mode.

Step#	σ_1	σ_2	σ_3	σ_4
0	$S_3 \ \iota_1 a n_2 n_3 n_4$	$S_3 \ \iota_2 n_1 n_3 n_4$	$S_3 \ \iota_3 n_1 n_2 n_4$	$S_3 \ \iota_4 n_1 n_2 n_3$
4	$S_4 \ \iota_1 a$	$S_4 \ \iota_2 p_1$	$S_4 \ \iota_3 p_1$	$S_4 \ \iota_4 p_1$

Table 12. Partial Trace of Algorithm 10 for Figure 2 in asynchronous mode.

Step#	σ_1	σ_2	σ_3	σ_4
0	$S_3 \ \iota_1 a n_2 n_3 n_4$	$S_3 \ \iota_2 n_1 n_3 n_4$	$S_3 \ \iota_3 n_1 n_2 n_4$	$S_3 \ \iota_4 n_1 n_2 n_3$
4	$S_4 \ \iota_1 a$	$S_4 \ \iota_2 p_1$	$S_4 \ \iota_3 p_2$	$S_4 \ \iota_4 p_3$

3. Rules in state S_3:
   1 $S_3\ n_j \rightarrow_{\texttt{min}} S_4\ w_j\ (c_i t) \downarrow_j |_{a\iota_i}$
   2 $S_3\ c_j n_j n_k \rightarrow_{\texttt{min}}$
   $S_4\ p_j w_k\ (c_i t) \downarrow_k |_{\iota_i}$
   3 $S_3\ c_j n_j \rightarrow_{\texttt{min}} S_4\ p_j\ (c_i t) \downarrow_j |_{\iota_i}$
   4 $S_3\ n_j \rightarrow_{\texttt{min}} S_4\ w_j\ (c_i t) \downarrow_j |_{t\iota_i}$
   5 $S_3\ t \rightarrow_{\texttt{max}} S_4$
4. Rules for state S_4:
   1 $S_4\ w_j \rightarrow_{\texttt{min}} S_4 |_{c_j}$
   2 $S_4\ w_j p_k \rightarrow_{\texttt{min}} S_4\ w_j p_k$
   3 $S_4\ w_j a \rightarrow_{\texttt{min}} S_4\ w_j a$
   4 $S_4\ c_j \rightarrow_{\texttt{min}} S_4$
   5 $S_4\ p_j \rightarrow_{\texttt{min}} S_4\ p_j\ (c_i t) \downarrow_j |_{t\iota_i}$
   6 $S_4\ t \rightarrow_{\texttt{max}} S_4$

6 Complexity

All our distributed DFS and BFS implementations, except the SoD implementation, assume that each cells knows the IDs of its neighbors (parents and children). Our SoD implementation assumes that each cell knows the labels of its adjacent arcs (incoming and outgoing). In the complexity analysis, we skip over a preliminary phase which could build such knowledge, see Algorithm 1.

All our P system DFS implementations take one final step, to prompt the source cell to discard the token; we also omit this step in the complexity analysis. Moreover, there is one beginning step in our implementations for Awerbuch (rule 3.1) and Cidon (rules 3.1, 3.2), which instantiates initial list objects. These steps can be included in Algorithm 1. However, we do not follow this approach, because we want to keep Algorithm 1 a common preliminary phase for all our algorithms. We also skip these beginning steps, in the complexity analysis.

Table 13 shows the resulting complexity of our P system DFS implementations, in terms of P steps. The runtime complexity of our P system implementations is exactly the same as for the standard distributed DFS algorithms. The complexity of our SOD algorithm must be considered with a big grain of salt, for the reasons explained in the description of Algorithm 7 (high-level pseudo-code).

Table 13. DFS algorithms comparisons and complexity (P steps) of Figure 3.

Algorithm	P Steps	Time units	Messages	Notes
Classical	18	$2M$	$2M$	Local cell IDs
Awerbuch	22	$4N-2$	$4M$	Local cell IDs
Cidon	10	$2N-2$	$\leq 4M$	Local cell IDs
Sharma	10	$2N-2$	$\leq 2N-2$	Global cell IDs
SOD	10?	$2N-2$	$\leq 2N-2$	Sense of Direction (Z_n)
Makki	9	$(1+r)N$	$(1+r)N$	Global cell IDs (or SOD)

Table 14 shows the runtime complexity of our P system SyncBFS and AsyncBFS implementations, which is consistent with the runtime complexity of the standard algorithms.

Table 14. BFS algorithms comparisons and complexity (P steps) of Figure 5.

Algorithm	P Steps	Time units	Messages	Notes
Sync	8	$O(D)$	$O(M)$	Local IDs
Simple Async	5	$O(DN)$	$O(NM)$	Local IDs
Simple Async2	?	$O(D^2)$	$O(DM)$	D and Local IDs
Layered Async	?	$O(D^2)$	$O(M + DN)$	Local IDs
Hybrid Async	?	$O(Dk + D^2/k)$	$O(Mk + DN/k)$	Local IDs

7 Conclusions

We proposed a new approach to fully asynchronous P systems, and a matching complexity measure, both inspired from the field of distributed algorithms. We validated our approach by implementing several well-known distributed depth-first search (DFS) and breadth-first search (BFS) algorithms. We believe that these are the first P implementations of the standard distributed DFS and BFS algorithms. Empirical results show that, in terms of P steps, the runtime complexity of our distributed P algorithms is the same as the runtime complexity of standard distributed DFS and BFS.

Several interesting questions remain open. We intend to complete this quest by completing the implementation of the SOD algorithm and by implementing three other, more sophisticated, distributed BFS algorithms and compare their performance against the standard versions. We also intend to elaborate the foundations of fully asynchronous P systems and further validate this, by investigating a few famous critical problems, such as building minimal spanning trees. Finally, we intend to formulate fundamental distributed asynchronous concepts, such as fairness, safety and liveness, and investigate methods for their proofs.

References

1. Awerbuch, B.: A new distributed depth-first-search algorithm. Information Processing Letters 20(3), 147 – 150 (1985), http://www.sciencedirect.com/science/article/B6V0F-482R9G2-S/2/22537b651ddd5c1a0e3ae5d5ba723079
2. Bernardini, F., Gheorghe, M., Margenstern, M., Verlan, S.: How to synchronize the activity of all components of a P system? Int. J. Found. Comput. Sci. 19(5), 1183–1198 (2008)
3. Casiraghi, G., Ferretti, C., Gallini, A., Mauri, G.: A membrane computing system mapped on an asynchronous, distributed computational environment. In: Freund, R., Paun, G., Rozenberg, G., Salomaa, A. (eds.) Workshop on Membrane Computing. Lecture Notes in Computer Science, vol. 3850, pp. 159–164. Springer (2005)
4. Cavaliere, M., Egecioglu, O., Ibarra, O., Ionescu, M., Pun, G., Woodworth, S.: Asynchronous spiking neural p systems: Decidability and undecidability. In: Garzon, M., Yan, H. (eds.) DNA Computing, Lecture Notes in Computer Science, vol. 4848, pp. 246–255. Springer Berlin / Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-77962-9_26

5. Cavaliere, M., Ibarra, O.H., Pun, G., Egecioglu, O., Ionescu, M., Woodworth, S.: Asynchronous spiking neural p systems. Theor. Comput. Sci. 410, 2352–2364 (May 2009), http://portal.acm.org/citation.cfm?id=1539070.1540146
6. Cavaliere, M., Sburlan, D.: Time and synchronization in membrane systems. Fundam. Inf. 64, 65–77 (July 2004), http://portal.acm.org/citation.cfm?id=1227085.1227092
7. Cidon, I.: Yet another distributed depth-first-search algorithm. Inf. Process. Lett. 26, 301–305 (1988)
8. Dinneen, M.J., Kim, Y.B., Nicolescu, R.: Edge- and node-disjoint paths in P systems. Electronic Proceedings in Theoretical Computer Science 40, 121–141 (2010)
9. Dinneen, M.J., Kim, Y.B., Nicolescu, R.: Edge- and vertex-disjoint paths in P modules. In: Ciobanu, G., Koutny, M. (eds.) Workshop on Membrane Computing and Biologically Inspired Process Calculi. pp. 117–136 (2010)
10. Dinneen, M.J., Kim, Y.B., Nicolescu, R.: P systems and the Byzantine agreement. Journal of Logic and Algebraic Programming 79(6), 334–349 (2010), http://www.sciencedirect.com/science/article/B6W8D-4YPPPW1-2/2/17b82b2cdd8f159b7fea380939193e4d
11. Freund, R.: Asynchronous p systems and p systems working in the sequential mode. In: Mauri, G., Paun, G., Prez-Jimnez, M., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 3365, pp. 36–62. Springer Berlin / Heidelberg (2005), http://dx.doi.org/10.1007/978-3-540-31837-8_3
12. Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J.: Depth-first search with p systems. In: Proceedings of the 11th international conference on Membrane computing. pp. 257–264. CMC'10, Springer-Verlag, Berlin, Heidelberg (2010), http://portal.acm.org/citation.cfm?id=1946067.1946090
13. Kleijn, J., Koutny, M.: Synchrony and asynchrony in membrane systems. In: Membrane Computing, WMC2006, Leiden, Revised, Selected and Invited Papers, LNCS 4361. pp. 66–85. Springer (2006)
14. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
15. Makki, S.A.M., Havas, G.: Distributed algorithms for depth-first search. Inf. Process. Lett. 60, 7–12 (October 1996), http://portal.acm.org/citation.cfm?id=244081.244085
16. Nicolescu, R., Dinneen, M.J., Kim, Y.B.: Discovering the membrane topology of hyperdag P systems. In: Păun, G., Pérez-Jiménez, M.J., Riscos-Núñez, A., Rozenberg, G., Salomaa, A. (eds.) Workshop on Membrane Computing. Lecture Notes in Computer Science, vol. 5957, pp. 410–435. Springer-Verlag (2009)
17. Nicolescu, R., Wu, H.: BFS solution for disjoint paths in P systems. Report CDMTCS-399, Centre for Discrete Mathematics and Theoretical Computer Science, The University of Auckland, Auckland, New Zealand (March 2011), http://www.cs.auckland.ac.nz/CDMTCS//researchreports/399radu.pdf
18. Pan, L., Zeng, X., Zhang, X.: Time-free spiking neural p systems. Neural Computation 0(0), 1–23 (2011), http://www.mitpressjournals.org/doi/abs/10.1162/NECO_a_00115
19. Păun, G.: Introduction to membrane computing. In: Ciobanu, G., Pérez-Jiménez, M.J., Păun, G. (eds.) Applications of Membrane Computing, pp. 1–42. Natural Computing Series, Springer-Verlag (2006)
20. Păun, G., Centre, T., Science, C.: Computing with membranes. Journal of Computer and System Sciences 61, 108–143 (1998)

21. Sharma, M.B., Iyengar, S.S.: An efficient distributed depth-first-search algorithm. Inf. Process. Lett. 32, 183–186 (September 1989), http://portal.acm.org/citation.cfm?id=69686.69691
22. Tel, G.: Introduction to Distributed Algorithms. Cambridge University Press (2000)
23. Tsin, Y.H.: Some remarks on distributed depth-first search. Inf. Process. Lett. 82, 173–178 (May 2002), http://portal.acm.org/citation.cfm?id=585580.585581
24. Yuan, Z., Zhang, Z.: Asynchronous spiking neural p system with promoters. In: Proceedings of the 7th international conference on Advanced parallel processing technologies. pp. 693–702. APPT'07, Springer-Verlag, Berlin, Heidelberg (2007), http://portal.acm.org/citation.cfm?id=1785246.1785331

A Appendix

Table 15. Awerbuch DFS algorithm traces (steps 0, ..., 15) of Figure 3 in synchronous mode, where σ_1 is the source cell.

Step	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	$S_0 a \iota_1$	$S_0 \iota_2$	$S_0 \iota_3$	$S_0 \iota_4$	$S_0 \iota_5$	$S_0 \iota_6$
1	$S_1 a \iota_1 y$	$S_0 \iota_2 z$	$S_0 \iota_3$	$S_0 \iota_4 z$	$S_0 \iota_5$	$S_0 \iota_6$
2	$S_2 a \iota_1 z^2$	$S_1 \iota_2 n_1 y z$	$S_0 \iota_3 z^2$	$S_1 \iota_4 n_1 y z$	$S_0 \iota_5 z$	$S_0 \iota_6$
3	$S_3 a \iota_1 n_2 n_4$	$S_2 \iota_2 n_1 n_4 z$	$S_1 \iota_3 n_2 n_4 y z$	$S_2 \iota_4 n_1 n_2 z^2$	$S_1 \iota_5 n_4 y z$	$S_0 \iota_6 z^2$
4	$S_4 a \iota_1 m_2 m_4 n_2 n_4$	$S_3 \iota_2 n_1 n_3 n_4$	$S_2 \iota_3 n_2 n_4 n_5 z$	$S_3 \iota_4 n_1 n_2 n_3 n_5$	$S_2 \iota_5 n_3 n_4 z$	$S_1 \iota_6 n_3 n_5 y$
5	$S_5 a \iota_1 m_2 m_4 n_2 n_4$	$S_4 \iota_2 m_1 m_3 m_4 n_1 n_3 n_4$ v_1	$S_3 \iota_3 n_2 n_4 n_5 n_6$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2$ $n_3 n_5 v_1$	$S_3 \iota_5 n_3 n_4 n_6$	$S_2 \iota_6 n_3 n_5$
6	$S_6 a b_2 b_4 \iota_1 m_2 m_4 n_2$ $n_4 w_2 w_4$	$S_4 \iota_2 m_1 m_3 m_4 n_1 n_3 n_4$ u_1	$S_4 \iota_3 m_2 m_4 m_5 m_6 n_2 n_4$ $n_5 n_6$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2$ $n_3 n_5 u_1$	$S_4 \iota_5 m_3 m_4 m_6 n_3 n_4 n_6$	$S_3 \iota_6 n_3 n_5$
7	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_4 c_1 \iota_2 m_1 m_3 m_4 n_1 n_3$ $n_4 t u_1$	$S_4 \iota_3 m_2 m_4 m_5 m_6 n_2 n_4$ $n_5 n_6$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2$ $n_3 n_5 u_1$	$S_4 \iota_5 m_3 m_4 m_6 n_3 n_4 n_6$	$S_4 \iota_6 m_3 m_5 n_3 n_5$
8	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_5 \iota_2 m_3 m_4 n_3 n_4 p_1 t\ u_1$	$S_4 \iota_3 m_2 m_4 m_5 m_6 n_2 n_4$ $n_5 n_6 v_2$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2$ $n_3 n_5 u_1 v_2$	$S_4 \iota_5 m_3 m_4 m_6 n_3 n_4 n_6$	$S_4 \iota_6 m_3 m_5 n_3 n_5$
9	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_6 b_3 b_4 \iota_2 m_3 m_4 n_3 n_4$ $p_1 t u_1 w_3 w_4$	$S_4 \iota_3 m_2 m_4 m_5 m_6 n_2 n_4$ $n_5 n_6 u_2$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2$ $n_3 n_5 u_1 u_2$	$S_4 \iota_5 m_3 m_4 m_6 n_3 n_4 n_6$	$S_4 \iota_6 m_3 m_5 n_3 n_5$
10	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_7 \iota_2 m_4 n_3 n_4 p_1 u_1 u_3$	$S_4 c_2 \iota_3 m_2 m_4 m_5 m_6 n_2$ $n_4 n_5 n_6 t u_2$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2$ $n_3 n_5 u_1 u_2$	$S_4 \iota_5 m_3 m_4 m_6 n_3 n_4 n_6$	$S_4 \iota_6 m_3 m_5 n_3 n_5$
11	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_7 \iota_2 m_4 n_3 n_4 p_1 u_1 u_3$	$S_5 \iota_3 m_4 m_5 m_6 n_4 n_5 n_6$ $p_2 t u_2$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2$ $n_3 n_5 u_1 u_2 v_3$	$S_4 \iota_5 m_3 m_4 m_6 n_3 n_4 n_6$ v_3	$S_4 \iota_6 m_3 m_5 n_3 n_5 v_3$
12	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_7 \iota_2 m_4 n_3 n_4 p_1 u_1 u_3$	$S_6 b_4 b_5 b_6 \iota_3 m_4 m_5 m_6$ $n_4 n_5 n_6 p_2 t u_2 w_4 w_5\ w_6$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2$ $n_3 n_5 u_1 u_2 u_3$	$S_4 \iota_5 m_3 m_4 m_6 n_3 n_4 n_6$ u_3	$S_4 \iota_6 m_3 m_5 n_3 n_5 u_3$
13	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_7 \iota_2 m_4 n_3 n_4 p_1 u_1 u_3$	$S_7 \iota_3 m_4 m_6 n_4 n_5 n_6 p_2$ $u_2 u_5$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2$ $n_3 n_5 u_1 u_2 u_3$	$S_4 c_3 \iota_5 m_3 m_4 m_6 n_3 n_4$ $n_6 t u_3$	$S_4 \iota_6 m_3 m_5 n_3 n_5 u_3$
14	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_7 \iota_2 m_4 n_3 n_4 p_1 u_1 u_3$	$S_7 \iota_3 m_4 m_6 n_4 n_5 n_6 p_2$ $u_2 u_5$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2$ $n_3 n_5 u_1 u_2 u_3 v_5$	$S_5 \iota_5 m_4 m_6 n_4 n_6 p_3 t\ u_3$	$S_4 \iota_6 m_3 m_5 n_3 n_5 u_3 v_5$
15	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_7 \iota_2 m_4 n_3 n_4 p_1 u_1 u_3$	$S_7 \iota_3 m_4 m_6 n_4 n_5 n_6 p_2$ $u_2 u_5$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2$ $n_3 n_5 u_1 u_2 u_3 u_5$	$S_6 b_4 b_6 \iota_5 m_4 m_6 n_4 n_6$ $p_3 t u_3 w_4 w_6$	$S_4 \iota_6 m_3 m_5 n_3 n_5 u_3 u_5$

Table 16. Awerbuch DFS algorithm traces (steps 16, ..., 27) of Figure 3 in synchronous mode, where σ_1 is the source cell.

Step	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
16	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_7 \iota_2 m_4 n_3 n_4 p_1 u_1 u_3$	$S_7 \iota_3 m_4 m_6 n_4 n_5 n_6 p_2 u_2 u_5$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2 n_3 n_5 u_1 u_2 u_3 u_5$	$S_7 \iota_5 m_4 n_4 n_6 p_3 u_3 u_6$	$S_4 c_5 \iota_6 m_3 m_5 n_3 n_5 t u_3 u_5$
17	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_7 \iota_2 m_4 n_3 n_4 p_1 u_1 u_3$	$S_7 \iota_3 m_4 m_6 n_4 n_5 n_6 p_2 u_2 u_5 v_6$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2 n_3 n_5 u_1 u_2 u_3 u_5$	$S_7 \iota_5 m_4 n_4 n_6 p_3 u_3 u_6$	$S_5 \iota_6 m_3 n_3 p_5 t u_3 u_5$
18	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_7 \iota_2 m_4 n_3 n_4 p_1 u_1 u_3$	$S_7 \iota_3 m_4 m_6 n_4 n_5 n_6 p_2 u_2 u_5 u_6$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2 n_3 n_5 u_1 u_2 u_3 u_5$	$S_7 \iota_5 m_4 n_4 n_6 p_3 u_3 u_6$	$S_6 b_3 \iota_6 m_3 n_3 p_5 t u_3 u_5 w_3$
19	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_7 \iota_2 m_4 n_3 n_4 p_1 u_1 u_3$	$S_7 \iota_3 m_4 n_4 n_5 n_6 p_2 u_2 u_5 u_6$	$S_4 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2 n_3 n_5 u_1 u_2 u_3 u_5$	$S_7 \iota_5 m_4 n_4 n_6 p_3 t u_3 u_6 x_6$	$S_7 \iota_6 n_3 p_5 u_3 u_5$
20	$S_7 a \iota_1 m_4 n_2 n_4 u_2$	$S_7 \iota_2 m_4 n_3 n_4 p_1 u_1 u_3$	$S_7 \iota_3 m_4 n_4 n_5 n_6 p_2 u_2 u_5 u_6$	$S_4 c_5 \iota_4 m_1 m_2 m_3 m_5 n_1 n_2 n_3 n_5 t u_1 u_2 u_3 u_5$	$S_7 \iota_5 n_4 n_6 p_3 u_3 u_4 u_6$	$S_7 \iota_6 n_3 p_5 u_3 u_5$
21	$S_7 a \iota_1 m_4 n_2 n_4 u_2 v_4$	$S_7 \iota_2 m_4 n_3 n_4 p_1 u_1 u_3\ v_4$	$S_7 \iota_3 m_4 n_4 n_5 n_6 p_2 u_2 u_5 u_6 v_4$	$S_5 \iota_4 m_1 m_2 m_3 n_1 n_2 n_3 p_5 t u_1 u_2 u_3 u_5$	$S_7 \iota_5 n_4 n_6 p_3 u_3 u_4 u_6$	$S_7 \iota_6 n_3 p_5 u_3 u_5$
22	$S_7 a \iota_1 m_4 n_2 n_4 u_2 u_4$	$S_7 \iota_2 m_4 n_3 n_4 p_1 u_1 u_3\ u_4$	$S_7 \iota_3 m_4 n_4 n_5 n_6 p_2 u_2 u_4 u_5 u_6$	$S_6 b_1 b_2 b_3 \iota_4 m_1 m_2 m_3 n_1 n_2 n_3 p_5 t u_1 u_2 u_3 u_5 w_1 w_2 w_3$	$S_7 \iota_5 n_4 n_6 p_3 u_3 u_4 u_6$	$S_7 \iota_6 n_3 p_5 u_3 u_5$
23	$S_7 a \iota_1 n_2 n_4 u_2 u_4$	$S_7 \iota_2 n_3 n_4 p_1 u_1 u_3 u_4$	$S_7 \iota_3 n_4 n_5 n_6 p_2 u_2 u_4 u_5 u_6$	$S_7 \iota_4 n_1 n_2 n_3 p_5 u_1 u_2 u_3 u_5$	$S_7 \iota_5 n_4 n_6 p_3 t u_3 u_4\ u_6 x_4$	$S_7 \iota_6 n_3 p_5 u_3 u_5$
24	$S_7 a \iota_1 n_2 n_4 u_2 u_4$	$S_7 \iota_2 n_3 n_4 p_1 u_1 u_3 u_4$	$S_7 \iota_3 n_4 n_5 n_6 p_2 t u_2 u_4 u_5 u_6 x_5$	$S_7 \iota_4 n_1 n_2 n_3 p_5 u_1 u_2 u_3 u_5$	$S_7 \iota_5 n_4 n_6 p_3 u_3 u_4 u_6$	$S_7 \iota_6 n_3 p_5 u_3 u_5$
25	$S_7 a \iota_1 n_2 n_4 u_2 u_4$	$S_7 \iota_2 n_3 n_4 p_1 t u_1 u_3\ u_4 x_3$	$S_7 \iota_3 n_4 n_5 n_6 p_2 u_2 u_4 u_5 u_6$	$S_7 \iota_4 n_1 n_2 n_3 p_5 u_1 u_2 u_3 u_5$	$S_7 \iota_5 n_4 n_6 p_3 u_3 u_4 u_6$	$S_7 \iota_6 n_3 p_5 u_3 u_5$
26	$S_7 a \iota_1 n_2 n_4 t u_2 u_4\ x_2$	$S_7 \iota_2 n_3 n_4 p_1 u_1 u_3 u_4$	$S_7 \iota_3 n_4 n_5 n_6 p_2 u_2 u_4 u_5 u_6$	$S_7 \iota_4 n_1 n_2 n_3 p_5 u_1 u_2 u_3 u_5$	$S_7 \iota_5 n_4 n_6 p_3 u_3 u_4 u_6$	$S_7 \iota_6 n_3 p_5 u_3 u_5$
27	$S_7 a \iota_1 n_2 n_4 u_2 u_4$	$S_7 \iota_2 n_3 n_4 p_1 u_1 u_3 u_4$	$S_7 \iota_3 n_4 n_5 n_6 p_2 u_2 u_4 u_5 u_6$	$S_7 \iota_4 n_1 n_2 n_3 p_5 u_1 u_2 u_3 u_5$	$S_7 \iota_5 n_4 n_6 p_3 u_3 u_4 u_6$	$S_7 \iota_6 n_3 p_5 u_3 u_5$