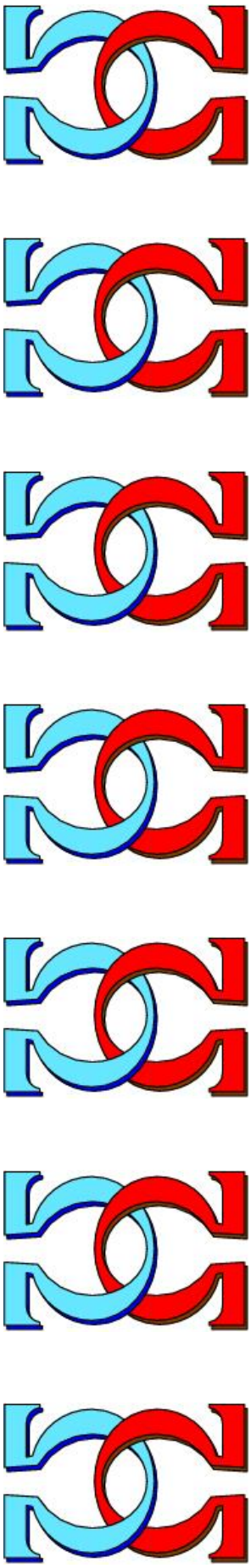

CDMTCS
Research
Report
Series

Fast Distributed DFS Solutions for Edge-disjoint Paths in Digraphs

Hossam ElGindy
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

Radu Nicolescu
Huiling Wu
Department of Computer Science, University of Auckland, Auckland, New Zealand

CDMTCS-431
December 2012

Centre for Discrete Mathematics and Theoretical Computer Science

Fast Distributed DFS Solutions for Edge-disjoint Paths in Digraph

Hossam ElGindy
School of Computer Science and Engineering,
University of New South Wales, Sydney, Australia
elgindyh@cse.unsw.edu.au

Radu Nicolescu and Huiling Wu
Department of Computer Science, University of Auckland,
Private Bag 92019, Auckland, New Zealand
r.nicolescu@auckland.ac.nz, hwu065@aucklanduni.ac.nz

December 10, 2012

Abstract

We present two new synchronous distributed message-based depth-first search (DFS) based algorithms, Algorithms C and D, to compute a maximum cardinality set of edge-disjoint paths, between a source node and a target node in a digraph. We compare these new algorithms with our previous implementation of the classical algorithm, Algorithm A, and our previous improvement, Algorithm B [10]. Empirical results show that, on a set of random digraphs, our algorithms are faster than the classical Algorithm A, by a factor around 40%. All these improved algorithms have been inspired and guided by a P system modelling exercise, but are suitable for any distributed implementation. To achieve the maximum theoretical performance, our P systems specification uses high-level generic rules applied in matrix grammar mode.

Keywords: edge-disjoint paths, depth-first search, network flow, distributed systems, P systems, generic rules, matrix grammars.

1 Introduction

The edge-disjoint paths problem finds a maximum cardinality set of edge-disjoint paths between a source node and a target node in a *digraph*. The classical algorithm transforms this problem to a maximum flow problem, solved by assigning unit capacity to each arc.

All algorithms discussed in this paper are distributed, totally message-based (no shared memory) and work synchronously: briefly, we call them *distributed*, implicitly assuming their other characteristics. In this paper, Algorithm A is a distributed version

of the classical edge-disjoint algorithm, based on Ford-Fulkerson's maximum flow algorithm [5] and the classical distributed DFS [13]. Algorithm A* is its slightly improved version, proposed by Dinneen et al. [3].

Recently, we proposed an improved distributed algorithm [10], here called Algorithm B. Algorithm B improved Algorithms A and A* by (a) using Cidon's distributed DFS [2, 13], which avoids revisiting cells in the same round, and (b) a *novel* idea, discarding "dead" cells detected during *failed* rounds, i.e. cells that will never appear in a successful search.

Here we propose two distributed algorithms: (1) Algorithm C, which, using a *different* idea, discards "dead" cells identified in *successful* and *failed* rounds, and (2) Algorithm D, which *combines* the benefits of Algorithms B and C.

Briefly, in all our algorithms, B, C, and D, search rounds explore *unvisited* cells and arcs. Cells and arcs encountered during the search are tentatively marked as *temporarily visited*. Temporarily visited cells and arcs which are detected "dead" are marked as *permanently visited* and ignored in the next search round. At the end of each search round, remaining *temporarily visited* cells and arcs revert to the *unvisited* status and can be revisited by next search round.

Our algorithms differ (1) in the rules used to detect "dead" cells and (2) in the process used to discard these "dead" cells for the next rounds. Our previous Algorithm B detects "dead" cells at the end of *failed* search rounds (only) and discards them in "real-time", on *shortest paths*. Our new Algorithm C can detect "dead" cells during any kind of search round (regardless if it is *failed* or *successful*) but discards these on the current *search path trace*, which is typically longer than the shortest path possible (especially in digraphs). In contrast, classical algorithms, such as Algorithms A and A*, do not discard any cell, and reset all cells as *unvisited*, at each search round end.

We also consider a *restricted* version of Algorithm C, called Algorithm C*, where we intentionally *omit* to discard "dead" cells found in *failed* rounds: in this sense, Algorithm C* is the opposite of Algorithm B. We can thus better assess the power of the main new idea behind Algorithm C: even its restricted version, C*, still *detects* a superset of all "dead" cells detected by Algorithm B. However, due to digraphs propagation delays, Algorithms C and C* are not always able to *prune* all detected cells in "real-time": they could prune all, if allowed to run longer. Thus, there are scenarios when one of Algorithms B and C is more suitable than the other. Algorithm D achieves maximum performance: it runs fast and detects and prunes all "dead" cells that can be detected by the combination of Algorithms B and C.

All these improved algorithms have been inspired and guided by a P system modelling exercise, but are suitable for any distributed implementation. A P system is a parallel and distributed computational model inspired by the structure and interactions of living cells, introduced by Păun [11]; for a recent overview of the domain, see Păun et al.'s recent monograph [12]. Essentially, a P system is specified by its membrane structure, symbols and rules. The underlying structure is a digraph or a more specialized version, such as a directed acyclic graph (dag) or a tree (which seems the most studied case). Each cell transforms its content symbols and sends messages to its neighbours using formal rules inspired by rewriting systems. Rules of the same cell can be applied in parallel (where possible) and all cells work in parallel, traditionally in the synchronous mode.

In this paper, we also assess P systems as directly executable formal specifications of synchronous distributed algorithms. Thus, we aim to construct P algorithms that compare favourably with high-level non-executable pseudocode: (1) first, in runtime complexity and (2) if possible, in program readability and size (which is independent of the problem size). Toward these goals, we use high-level generic P rules, applied using a new proposed semantics, inspired from *matrix* grammars. Our previous algorithms have used a related, but less powerful, application mode, the so-called weak priority mode. The weak priority mode seems adequate for simple algorithms, but the novel matrix semantics is more suitable for more sophisticated algorithms, such as our new algorithms presented here.

2 Edge-disjoint Paths in Digraphs

We consider a *digraph*, $G = (V, E)$, where V is a finite set of *nodes*, $V = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, and E is a set of *arcs*. For consistency with the P system terminologies, the *nodes* of V are also called *cells*. Digraph arcs define (*parent*, *child*) relationships, e.g., arc $(\sigma_i, \sigma_j) \in E$ defines σ_j as σ_i's child and σ_i as σ_j's parent; with alternate notations, $\sigma_j \in E(\sigma_i)$, $\sigma_i \in E^{-1}(\sigma_j)$. A *path* is a finite ordered set of nodes successively connected by arcs. A *simple path* is a path with no repeated nodes. Clearly, any path can be "streamlined" to a simple path, by removing repeated nodes. Given a path, π, we define: $\overline{\pi} \subseteq E$, as the set of its arcs and its reversal, $\overline{\pi}^{-1} = \{(\sigma_j, \sigma_i) \mid (\sigma_i, \sigma_j) \in \overline{\pi}\} \subseteq E^{-1}$.

Given a *source* node, $s \in V$, and a *target* node, $t \in V$, the edge-disjoint problem looks for a *maximum cardinality set* of edge-disjoint s-to-t paths. A set of paths are edge-disjoint if they have no common arc. If the edge-disjoint paths are not *simple*, we can always *simplify* them at the end. The edge-disjoint problem can be transformed to a maximum flow problem, by assigning unit capacity to each arc [8].

Given a set of edge-disjoint paths P, we define $\overline{P}$ as the set of their arcs, $\overline{P} = \cup_{\pi \in P} \overline{\pi}$, and the *residual* digraph $G_P = (V, E_P)$, where $E_P = (E \setminus \overline{P}) \cup \overline{P}^{-1}$. Briefly, the residual digraph is constructed by reversing arcs in $\overline{P}$.

Given a set of edge-disjoint paths, P, an *augmenting* path, α, is an s-to-t path in G_P. Augmenting paths are used to extend an already established set of edge-disjoint paths. An augmenting path arc is either (1) an arc in $E \setminus \overline{P}$ or (2) an arc in $\overline{P}^{-1}$, i.e. it reverses an existing arc in $\overline{P}$. Case (2) is known as a *push-back* operation: when it occurs, the arc in $\overline{P}$ and its reversal in $\overline{\alpha}$ "cancel" each other and are discarded. The remaining path fragments are relinked to construct an extended set of edge-disjoint paths, P', where $\overline{P'} = (\overline{P} \setminus \overline{\alpha}^{-1}) \cup (\overline{\alpha} \setminus \overline{P}^{-1})$. This process is repeated, starting with the new and larger set of edge-disjoint paths, P', until no more augmenting paths are found [5].

Figure 1 shows how to find an augmenting path in a residual digraph: (a) shows the initial digraph, G, with two edge-disjoint paths, $P = \{\sigma_0.\sigma_1.\sigma_4.\sigma_7,\ \sigma_0.\sigma_2.\sigma_5.\sigma_7\}$; (b) shows the residual digraph, G_P, formed by reversing edge-disjoint path arcs; (c) shows an augmenting path, $\alpha = \sigma_0.\sigma_3.\sigma_5.\sigma_2.\sigma_6.\sigma_7$, which uses a reverse arc, (σ_5, σ_2); (d) discards the cancelling arcs, (σ_2, σ_5) and (σ_5, σ_2); (e) relinks the remaining path fragments, $\sigma_0.\sigma_1.\sigma_4.\sigma_7$, $\sigma_0.\sigma_2$, $\sigma_5.\sigma_7$, $\sigma_0.\sigma_3.\sigma_5$ and $\sigma_2.\sigma_6.\sigma_7$, resulting in an incremented

set of three edge-disjoint paths, $P' = \{\sigma_0.\sigma_1.\sigma_4.\sigma_7,\ \sigma_0.\sigma_2.\sigma_6.\sigma_7,\ \sigma_0.\sigma_3.\sigma_5.\sigma_7\}$; (f) shows the new residual digraph, $G_{P'}$.

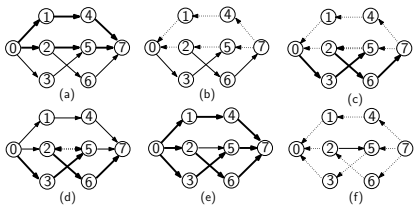

Figure 1: Finding an augmenting path in a residual digraph. Thin arcs: original arcs; thick arcs: disjoint or augmenting path arcs; dotted arcs: reversed path arcs.

Augmenting paths can be repeatedly searched using a DFS algorithm on *residual digraphs* [5], which dynamically builds *DFS trees*. A *search path*, τ, is a path, which starts from the source and "tries" to reach the target. A search path explores as far as possible before backtracking. At any given time, a search path is, either (1) a branch in the DFS tree or a prefix of it or (2) a branch in the DFS tree followed by one more arc, which, in a failed attempt, visits another node of the same branch or of another branch.

Our algorithms use a *synchronous* version of Cidon's *distributed* DFS [2, 13], which avoids case (2) above. When a node is first visited, it immediately marks all incoming arcs as visited, by sending *visited notifications* to its digraph parents. These notifications run in *parallel* with the main search, without delaying it. All parents are thus timely notified and, if they become visited, will not send the visiting token to this already visited node. For example, in Figure 2 (a), search path $\sigma_0.\sigma_1.\sigma_2.\sigma_3.\sigma_4.\sigma_6$ does not revisit cell σ_3. This is a powerful optimisation, which reduces the DFS complexity from $O(m)$ to $O(n)$; we use it, but this is not intrinsically related to our novel proposal.

When τ cannot explore further, it *backtracks*. The search is *successful* when the search path reaches the target. A successful search path becomes a new *augmenting path* and is used to increase the number of edge-disjoint paths: while conceptually a distinct operation, the new edge-disjoint paths are typically formed while the successful search path returns on its steps, back to the source (this successful return is distinct from the backtrack).

Given a *current* search path arc, (σ_i, σ_j), σ_i is a search path predecessor (*sp-predecessor*) of σ_j, and σ_j is a search path successor (*sp-successor*) of σ_i. Given a *previous* search path arc, (σ_i, σ_j), σ_i is a search tree predecessor (*st-predecessor*) of σ_j, and σ_j is a search tree successor (*st-successor*) of σ_i. Until the end of current search round, these arcs are considered *temporary visited*. At the end of the round, for the next search round, these arcs may become *permanently visited* or revert to *unvisited*.

In this paper, we propose a *novel* procedure to detect "dead" nodes, based on two numerical search-specific attributes: the (known) node *depth* and a new attribute which we call *reach-number*. A node's *depth*, σ_i.`depth`, is the number of hops from the source to itself in the search tree. A node's *reach-number* is the minimum of its depth and all its st-successors' reach-numbers; more precisely, it is the *greatest fixed point* that satisfies the following recursive equation:

$$\sigma_i.\texttt{reach} = \min(\{\sigma_i.\texttt{depth}\} \cup \{\sigma_j.\texttt{reach} \mid (\sigma_i, \sigma_j) \in E'\})$$

where E' is the current residual arcs set and assuming that *discarded* nodes have *infinite* reach-numbers.

As algorithmically determined, depths and reach-numbers start as *infinite* and are further *iteratively* adjusted during the search process:

1. When the search path first *visits* node σ_i:
   (a) both σ_i's depth and reach-number are set to the current hop count (see 4.7).
2. When the search path *backtracks* from node σ_k to node σ_i:
   (a) σ_i can *decrease* its reach-number, if $\sigma_k.\texttt{reach} < \sigma_i.\texttt{reach}$ (see 4.19);
   (b) σ_k can be *discarded*, if $\sigma_k.\texttt{reach} > \sigma_i.\texttt{depth}$ (see 4.20).
3. When node σ_i is *discarded*:
   (a) $\sigma_i.\texttt{reach}$ becomes *infinite* (see 5.6);
   (b) σ_i's st-predecessors can *increase* their reach-numbers (see 5.9 and 6.7);
   (c) all σ_i's st-successors can be discarded (this is a recursive procedure, see 5.14).
4. When node σ_j's reach-number is *increased* without being discarded, because its st-successor σ_i has increased its own reach-number:
   (a) σ_j's st-predecessors can also *increase* their reach-numbers (see 6.12);
   (b) σ_i can be *discarded*, if $\sigma_i.\texttt{reach} > \sigma_j.\texttt{depth}$ (see 6.14).

Note that, if finite, a node's reach-number is never greater than its own depth, i.e. $\sigma_i.\texttt{reach} \leq \sigma_i.\texttt{depth}$. Note also the two cases where a node can be discarded, (2.b) and (4.b), require a similar additional condition: this node's reach-number is changed to a finite number greater than all its parent's depths.

While items (1.a), (2.a) and (3.a) can be easily incorporated in any search, in a message-based distributed algorithm, items (2.b), (3.b), (4.a) and (4.b) must be recursively propagated by *notification messages*, over existing arcs. However, this is a *residual* digraph, where some residual arcs are inverted original arcs, some residual parents are original children and some residual children are original parents. The actual algorithm needs additional housekeeping to properly send such notifications, only to all concerned neighbours. Note that unvisited or discarded cells do not need these notifications.

These notifications travel in *parallel* with the main search activities, without affecting the overall performance. However, these notifications only travel along *search paths traces*, which, in digraphs, are *not* the *shortest* possible paths. Therefore, as we will see in a later example, not all cells can be effectively notified in "real-time", and may be reached by the next search process *before* they get their due discard or update notifications.

Briefly, in a digraph based system, we have a *pruning propagation delay*, which may negatively affect its performance.

As we see in Section 3, in Algorithm C, cases (2.b) and (4.b) trigger *discard notifications*, which are propagated by the function Discard and cases (3.b) and (4.a) trigger *update notifications*, propagated by the function Update.

Figure 2 illustrates how the depth and reach-numbers are initially set during forward moves and dynamically adjusted (decreased) during backtrack moves. In (a), $\sigma_0.\sigma_1.\sigma_2.\sigma_3.\sigma_4.\sigma_5.\sigma_3$ is a search path attempting to visit the already visited node σ_3 (if we use Cidon's optimisation, it will not actually visit σ_3). At this step, the reach-number of each node on the search path is still the same as its depth, $\sigma_i.\texttt{reach} = \sigma_i.\texttt{depth} = i, i \in [0, 5]$. A few steps later, in (b), the search path, $\sigma_0.\sigma_1.\sigma_2.\sigma_3$, has backtracked to σ_3. Cells to which we have backtracked, σ_5, σ_4 and σ_3, have updated their reach-numbers: $\sigma_5.\texttt{reach} = \min(\sigma_5.\texttt{depth}, \sigma_3.\texttt{reach}) = 3$; $\sigma_4.\texttt{reach} = \min(\sigma_4.\texttt{depth}, \sigma_6.\texttt{reach}) = 3$ and $\sigma_3.\texttt{reach} = \min(\sigma_3.\texttt{depth}, \sigma_4.\texttt{reach}) = 3$. After one more step, in (c), the search path, $\sigma_0.\sigma_1.\sigma_2.\sigma_3.\sigma_6$, moves forward to σ_6. At this step, $\sigma_6.\texttt{reach} = \sigma_6.\texttt{depth} = 4$. After one more step, in (d), the search path, $\sigma_0.\sigma_1.\sigma_2.\sigma_3$, backtracks again to σ_3 and $\sigma_3.\texttt{reach} = \min(\sigma_3.\texttt{depth}, \sigma_4.\texttt{reach}, \sigma_6.\texttt{reach}) = 3$ (unchanged). At this stage, we *discard* σ_6, because $\sigma_6.\texttt{reach} = 4 > \sigma_3.\texttt{depth} = 3$. One step later, in (e), the search path, $\sigma_0.\sigma_1.\sigma_2$, has backtracked to σ_2, and $\sigma_2.\texttt{reach} = \min(\sigma_2.\texttt{depth}, \sigma_3.\texttt{reach}) = 2$ (unchanged). We can now *discard* σ_3, σ_4 and σ_5, because their reach-numbers are greater than $\sigma_2.\texttt{depth} = 2$.

Another step later, the search will *succeed*, the search path, $\sigma_0.\sigma_1.\sigma_2.\sigma_7$, will reach σ_7 and become an augmenting path. The *discarded* cells, σ_3, σ_4, σ_5 and σ_6, can remain *permanently visited* and need not be further reconsidered. Conceptually, subsequent searches will use a trimmed digraph, which will speed up the algorithm. Our previous Algorithm B [10] does not discard these cells, because it uses a different idea, which only detects "dead" cells in *failed* searches.

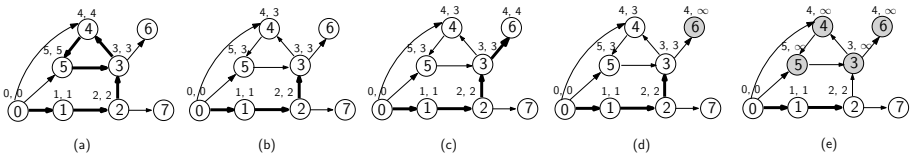

Figure 2: Thin arcs: original arcs; thick arcs: search path arcs; (depth, reach) pairs beside each node indicate the node's depth and reach-number; gray cells have been discarded.

3 High-level Pseudocode

Algorithm A is a distributed version of the classical Ford-Fulkerson based edge-disjoint paths algorithm [5]. To find augmenting paths, it uses the classical DFS algorithm [13]. This algorithm uses a **repeat-until** loop, repeatedly probing all unvisited children (both previously unprobed children and previously probed but failed children), resetting all

visited nodes and arcs as unvisited after each new augmenting path, until no more augmenting paths are found. Pseudocode 1 shows its high-level description, after unrolling its first DFS call.

To improve the readability, the pseudocode of this distributed algorithm and of all other discussed algorithms are presented in *sequentialized* versions. Each boxed area wraps code which is *essential* in a parallel run, but is *omitted* in the sequential mode. The **fork** keyword indicates the start of a *parallel* execution.

Algorithm A is described by Pseudocodes 1 and 2, which use the following variables: $G = (V, E)$ is the underlying digraph; $\sigma_s \in V$ is the source cell; $\sigma_t \in V$ is the target cell; r is the current round number; P_{r-1} is the set of edge-disjoint paths available at the start of round $\#r$; $G_{r-1} = (V, E_{r-1})$ is the residual digraph available at the start of round $\#r$. Algorithm A starts with an empty set of edge-disjoint paths, $P_0 = \emptyset$ and the trivial residual graph, $G_0 = (V, E_0)$, where $E_0 = E$ (i.e. $G_0 = G$).

Pseudocode 1: Algorithm A

```
1  Input: a digraph G = (V,E), a source cell, σ_s ∈ V,
2         and a target cell, σ_t ∈ V
3  r = 0, P_0 = ∅, G_0 = G
4  repeat
5    α = null
6    β = null
7    while there is an unvisited arc (σ_s, σ_q) ∈ E_{r-1} and β = null
8      r = r + 1
9      set σ_s and (σ_s, σ_q) as visited
10     β = DFS(σ_q, σ_t, G_{r-1})          // see Pseudocode 2
11     if β = null then                    // failed round
12       G_r = G_{r-1}
13     endif
14   endwhile
15   if β ≠ null then                      // successful round
16     α = σ_s.β
17     P_r = (P_{r-1} \ α^{-1}) ∪ (α \ P_{r-1}^{-1})
18     G_r = (V, E_r), where E_r = (E \ P_r) ∪ P_r^{-1}
19     reset all visited cells and arcs to unvisited
20   endif
21 until α = null
22 Output: P_r, which is a maximum cardinality set of edge-disjoint paths
```

Pseudocode 2: Classical DFS, for residual digraph $\mathbf{G_{r-1} = (V, E_{r-1})}$

```
1  DFS(σ_i, σ_t, G_{r-1})
2  Input: the current cell, σ_i ∈ V, the target cell, σ_t ∈ V
3         and the residual digraph, G_{r-1}
4  if σ_i = σ_t then return σ_t
5  if σ_i is visited then return null
6  set σ_i as visited
7  foreach unvisited (σ_i, σ_k) ∈ E_{r-1}
8    set (σ_i, σ_k) as visited
9    β = DFS(σ_k, σ_t, G_{r-1})
10   if β ≠ null return σ_i.β
11 endfor
```

12 **return null**
13 **Output**: a σ_i-to-σ_t path, if any; otherwise, **null**

Our new Algorithm C is described by Pseudocodes 3–6 and uses the same variables as Algorithm A; additionally, this algorithm works in d successive *search rounds*, defined by successive *iterations* of its **for** loop (line 3.8), where d is the outdegree of σ_s. Without loss of generality, we assume that σ_s's children are represented by the set $\{\sigma_{s1}, \sigma_{s2}, \ldots, \sigma_{sd}\}$.

Pseudocode 3: Algorithm C

1 **Input**: a digraph $G = (V, E)$, a source cell, $\sigma_s \in V$,
2 and a target cell, $\sigma_t \in V$
3 $P_0 = \emptyset$, $G_0 = G$
4 **set** σ_s as *permanently visited*
5 **foreach** *unvisited* arc $(\sigma_j, \sigma_s) \in E$
6 **set** (σ_j, σ_s) as *permanently visited*
7 **endfor**
8 **for** $r = 1$ **to** d
9 **if** σ_{sr} is *permanently visited* **then continue**
10 **set** (σ_s, σ_{sr}) as *permanently visited*
11 $\beta =$ NW_DFS$(\sigma_{sr}, \sigma_t, G_{r-1}, 1)$ // see Pseudocode 4
12 **if** $\beta =$ **null then** // failed round
13 **fork** Discard(σ_{sr}, G_{r-1})
14 $G_r = G_{r-1}$
15 **reset** all *temporarily visited* cells and arcs to *unvisited*
16 **else** // successful round
17 $\alpha = \sigma_s.\beta$
18 $\overline{P_r} = (\overline{P_{r-1}} \setminus \overline{\alpha}^{-1}) \cup (\overline{\alpha} \setminus \overline{P_{r-1}}^{-1})$
19 $G_r = (V, E_r)$, where $E_r = (E \setminus \overline{P_r}) \cup \overline{P_r}^{-1}$
20 **reset** all *temporarily visited* cells and arcs to *unvisited*
21 **endif**
22 **endfor**
23 **Output**: P_r, which is a maximum cardinality set of edge-disjoint paths

Pseudocode 4: NW-DFS, adapted for $\mathbf{G_{r-1}} = (\mathbf{V}, \mathbf{E_{r-1}})$

1 NW_DFS$(\sigma_i, \sigma_t, G_{r-1}, depth)$
2 **Input**: the current cell, $\sigma_i \in V$, the target cell, $\sigma_t \in V$,
3 the residual digraph, G_{r-1}, and σ_i's depth, *depth*
4 **if** $\sigma_i = \sigma_t$ **then return** σ_t
5 **if** σ_i is *permanently visited* **then return null**
6 **set** σ_i as *temporarily visited*
7 σ_i.reach $= \sigma_i$.depth $= depth$
8 **foreach** *unvisited* arc $(\sigma_j, \sigma_i) \in E_{r-1}$ // see Cidon's DFS
9 **set** (σ_j, σ_i) as *temporarily visited*
10 **endfor**
11 **foreach** arc $(\sigma_i, \sigma_k) \in E_{r-1}$
12 **if** (σ_i, σ_k) is *permanently visited* **then continue**
13 **elseif** (σ_i, σ_k) is *temporarily visited* **then**
14 σ_i.reach $= \min(\sigma_i$.reach, σ_k.reach$)$
15 **else** // unvisited
16 **set** (σ_i, σ_k) as *temporarily visited*
17 $\beta =$ NW_DFS$(\sigma_k, \sigma_t, G_{r-1}, depth + 1)$

```
18        if β = null then
19          σ_i.reach = min(σ_i.reach, σ_k.reach)
20          if σ_k.reach > σ_i.depth then [fork] Discard(σ_k, G_{r−1}) // see PC 5
21        else
22          return σ_i.β
23        endif
24      endif
25    endfor
26    return null
27    Output: a σ_i-to-σ_t path, if any; otherwise, null
```

Pseudocode 5: Discard, adapted for $\mathbf{G_{r-1} = (V, E_{r-1})}$

```
1   Discard(σ_i, G_{r−1})
2   Input: a cell to discard, σ_i ∈ V, and the residual digraph, G_{r−1}
3   if σ_i is permanently visited then return
4   set σ_i as permanently visited
5   ioldreach = σ_i.reach
6   σ_i.reach = ∞
7   foreach arc (σ_j, σ_i) ∈ E_{r−1}
8     if (σ_j, σ_i) is temporarily visited then
9       [fork] Update(σ_j, G_{r−1}, ioldreach, σ_i)        // see Pseudocode 6
10    endif
11    set (σ_j, σ_i) as permanently visited
12  endfor
13  foreach temporarily visited arc (σ_i, σ_k)
14    [fork] Discard(σ_k, G_{r−1})
15  endfor
```

Pseudocode 6: Update, adapted for $\mathbf{G_{r-1} = (V, E_{r-1})}$

```
1   Update(σ_j, G_{r−1}, ioldreach, σ_i)
2   Input: a cell, σ_j ∈ V, the residual digraph, G_{r−1}, a reach-number, ioldreach,
3          and a cell, σ_i ∈ V
4   if σ_j.reach = ioldreach then
5     newreach = σ_j.depth
6     foreach temporarily visited arc (σ_j, σ_k) ∈ E_{r−1}
7       newreach = min(newreach, σ_k.reach)
8     endfor
9     if newreach > σ_j.reach then
10      joldreach = σ_j.reach
11      σ_j.reach = newreach
12      [ foreach temporarily visited arc (σ_k, σ_j) ∈ E_{r−1}
          fork Update(σ_k, G_{r−1}, joldreach, σ_j)
        endfor ]
13    endif
14    [ if σ_i.reach > σ_j.depth then
        fork Discard(σ_i, G_{r−1})
      endif ]
15  endif
```

Search round #r starts when cell σ_s sends the *forward token*, together with a depth indication of one (lines 3.8–11), to unvisited cell σ_{sr}. A receiving cell marks itself as

temporarily visited (line 4.6), records its depth and reach-number as the received depth (line 4.7) and sends *visited notification* to all its neighbours (lines 4.8–10), becoming the new frontier cell. A current frontier cell sends the forward token over an arbitrarily selected unvisited arc, together with an incremented depth (lines 4.15–17), then the frontier advances. The visited notification housekeeping is performed in parallel with the main search. A frontier cell, which does not have any (more) unvisited arc, sends back a *backtrack token* to its search path predecessor, to return the frontier.

The search path backtracks to cell σ_i when: (1) σ_i avoids revisiting a temporarily visited child, σ_k (we consider this is a backtrack from σ_k, as in the classical DFS); (2) σ_i receives a backtrack token from σ_k. In both cases, σ_i recomputes its reach-number (lines 4.7, 4.13–15, 4.18–19). If σ_k's reach-number is greater than σ_i's depth, then σ_i sends a *discarding notification* to σ_k (line 4.20).

On receiving a discarding notification, cell σ_i sets itself as *permanently visited* and sets its reach-number as infinite (lines 5.4, 5.6). Also, it sends a *permanently visited notification* and an *update* to its st-predecessors (lines 5.7–5.12), σ_j's. On receiving an update from σ_i, cell σ_j recomputes its reach-number (lines 6.4–11). If σ_j increases its reach-number, it records and sends an update to its st-predecessors, σ_k's (line 6.12). If σ_i's reach-number is greater than σ_j's depth, then σ_j sends a discarding notification to σ_i (line 6.14).

Once cell σ_i is discarded, it sends discarding notifications to its st-successors (lines 5.13–15). This notification travels in parallel with the main search, along search path traces (not the shortest possible path). If a cell is not notified in "real time", it may be visited by the next search process before it gets its due discarding notification. To solve this problem, once discarded, the cell immediately backtracks (line 4.5) and sends an update to its st-predecessors (lines 5.7–12).

If the search path reaches the *target* cell, σ_t (line 4.4), then round $\#r$ succeeds and σ_t sends a *path confirmation* back to σ_s. While moving towards σ_s, the confirmation reshapes the existing edge-disjoint paths and the newly found augmenting path, α_r, building a larger set of edge-disjoint paths, and a new residual digraph. Thus, lines 3.17–19 of Pseudocode 3 are actually done within Pseudocode 4, during the return from a successful search. After receiving the path confirmation, σ_s initiates a global *reset*, which changes all temporarily visited cells and arcs to unvisited (line 3.20). This reset runs two steps ahead and in parallel with the next round, without affecting it.

If the search path cannot reach σ_t, the source, σ_s, receives a backtrack token from σ_{sr} and the round fails (line 3.12). Then σ_s sends a discarding notification to σ_{sr} (line 3.13) and initiates a global reset, to change all temporarily visited cells and arcs to unvisited (line 3.15). A failed round does not change the current set of edge-disjoint paths or the current residual digraph (line 3.14).

Although not explicit in the pseudocode, after probing all its children, the source, σ_s, initiates a global finalisation. This is not strictly necessary, but informs all cells that the algorithm has terminated.

Algorithm D combines Algorithms B and C, which discards all "dead" cells that are detected by B and C. Its Pseudocode 7 only changes one line of Pseudocode 3. Algorithm D uses two end-of-round resets, as in Algorithm B: (1) *after-success* reset, which resets all *temporarily visited* cells and arcs to *unvisited*, and (2) *after-failure* reset,

which sets all *temporarily visited* cells and arcs as *permanently visited* (to be discarded).

Pseudocode 7: Algorithm D
Same as Pseudocode 3, except line 15 is changed as follows:

15 **set** all *temporarily visited* cells and arcs to *permanently visited*

4 P Systems

We use a refined version of the *simple P module*, as defined in [4], where all cells share the same state and rule sets, extended with generic rules using complex symbols [9] and *matrix* organized rules (proposed here).

Definition 1 (Simple P module). A simple P module with duplex channels is a system $\Pi = (V, E, Q, O, R)$, where V is a finite set of cells; E is a set of structural *parent-child digraph* arcs between cells (functioning as *duplex* channels); Q is a finite set of states; O is a finite non-empty alphabet of symbols; and R is a finite set of multiset rewriting rules (further organized, as described below, in a matrix format).

In this paper, all components of a P module, i.e. V, E, Q, O and R, are immutable. Each cell, $\sigma_i \in V$, has the initial configuration (S_{i0}, w_{i0}), and the current configuration (S_i, w_i), where $S_{i0} \in Q$ is the initial state; $S_i \in Q$ is the current state; $w_{i0} \in O^*$ is the initial multiset of symbols; and $w_i \in O^*$ is the current multiset of symbols. The general form of a rule in R is:

$$S\ x \rightarrow_\alpha S'\ x'\ (y)\beta_\gamma \ldots \mid z \neg z',$$

where: $S, S' \in Q$, $x, x', y, z, z' \in O^*$, $\alpha \in \{\texttt{min}, \texttt{max}\}$, $\beta \in \{\uparrow, \downarrow, \updownarrow\}$, $\gamma \in V \cup \{\forall\}$ and ellipses $(\ldots)$ indicate possible repetitions of the last parenthesized item; state S is known as the rule's *starting* state and state S' as its *target* state.

For cell σ_i in configuration (S_i, w_i), a rule $S\ x \rightarrow_\alpha S'\ x'\ (y)\beta_\gamma \ldots \mid z \neg z' \in R$ is applicable if $S = S_i$, $xz \subseteq w_i$, $z' \cap w_i = \emptyset$ and either (a) no other rule was previously applied, in the same step, or (b) all rules previously applied, in the same step, have indicated the same target state, S'. When applied, this rule consumes multiset x and fixes, if not already fixed, the target state to S'. Multiset x', also known as the "here" multiset, remains in the same cell; in our *matrix* inspired formalism, x' becomes *immediately* available to other rules subsequently applied in the same step. Multiset y is a message *queued* and sent, at the end of the current step, as indicated by the transfer operator β_γ. β's arrow indicates the transfer direction: $\uparrow$—to parents; $\downarrow$—to children; $\updownarrow$—in both directions. γ indicates the distribution form: $\forall$—a broadcast; a *structural neighbour*, $\sigma_j \in V$—a unicast (to this neighbour). Multiset z is a *promoter* and z' is an *inhibitor*, which enables and disables the rule, respectively, without being consumed [12]. Operator α describes the rewriting mode. In the *minimal* mode, an applicable rule is applied once. In the *maximal* mode, an applicable rule is applied as many times as possible.

Matrix structured rulesets: We use matrix structured rulesets, which are inspired by matrix grammars with appearance checking [6]. Ruleset R is organized as a *matrix*,

i.e. a list of *vectors*: $R = (R_1, \ldots, R_m)$, $1 \leq m$, where vectors are listed from high-to-low priorities; all rules in a vector *share* the same *starting state*. Each vector R_i is a sequence of rules, $R_i = (R_{i,1}, \ldots, R_{i,m_i})$, $1 \leq m_i$, where rules are listed from high-to-low priorities. The matrix semantics combines a *strong priority* for vectors and a version of *weak priority* for rules inside a vector. Pseudocode 8 shows how ruleset R is applied.

A vector is applicable if at least one of its rules is applicable. In a given vector, R_i, rules are considered for application according to their (weak-like) priority order: (a) if applicable, a higher priority rule is applied before considering the next lower priority rule; (b) otherwise (if not applicable), a higher priority rule is silently ignored and the next priority rule is considered.

After a rule is applied, "here" symbols become *immediately available* for the next rule (in the same vector), while outgoing messages are queued until the end of the step (until all rules in the vector are considered). This is the difference with the classical weak priority rule, where "here" symbols do not become available until the end of the step, thus cannot be used by the next priority rule.

Vectors are considered for application in their (strong) priority order: (a) if applicable, a higher priority vector is applied and all lower priority vectors are ignored (for the current step); (b) otherwise (if not applicable), a higher priority vector is silently ignored and the next priority vector is considered. A *step* ends (1) after the application of the highest priority applicable vector, if any (this is an *active* step) or (2) when no vector is applicable (this is an *idle* step).

As a special case, the cell *stops* when it enters a state with no associated vectors, also known as a *final state*. Under this convention, a P system algorithm terminates after all cells enter a final state.

Pseudocode 8: Matrix structured ruleset application

```
1  Input : a P module, Π = (V, E, Q, O, R)
2          R = (R_1, ... R_m), 1 ≤ m, and R_i = (R_{i,1}, ..., R_{i,m_i}), 1 ≤ m_i
3  applied = false
4  for i = 1 to m
5     for j = 1 to m_i
6        if R_{i,j} is applicable then
7           apply R_{i,j}: "here" symbols become immediately available
8                          outgoing messages are queued
9           applied = true
10       endif
11    endfor
12    if applied then
13       send all queued messages
14       break
15    endif
16 endfor
```

Complex symbols: While atomic symbols seem sufficient for many theoretical studies (e.g, computational completeness), complex algorithms need adequate complex data structures. We enhance our initial vocabulary, by recursive composition of *elementary symbols* from O into *complex symbols*, which are compound terms of the form: $t(i, \ldots)$, where (1) t is an *elementary symbol* representing the functor; (2) i can be (a) an *ele-*

mentary symbol, (b) another *complex* symbol, (c) a *free variable* (open to be bound, according to the cell's current configuration), (d) a *multiset* of elementary and complex symbols and free variables.

We often abbreviate complex symbols by using subscripts for term arguments. The following are examples of complex symbols, where a, b, c, d, e, f are elementary symbols and i, j, X are free variables (assuming that these are not listed among elementary symbols): $b(2) = b_2$, $c(i) = c_i$, $d(i, j) = d_{i,j}$, $e(a^2b^3)$, $f(j, c^5) = f_j(c^5)$, $f(j, Xc) = f_j(Xc)$.

Besides modelling complex data structures, such as lists, stacks, trees and dictionaries, or emulating procedure calls, complex symbols are useful for representing and processing any number of cell IDs with a fixed vocabulary. Thus, complex symbols allow the design of *fixed-size* P system algorithms, i.e. algorithms having a fixed number of rules, which does not depend on the number of cells in the underlying P systems.

Here we assume that each cell σ_i is "blessed" with a unique complex *cell ID* symbol, $\iota(i)$, typically abbreviated as ι_i, which is exclusively used as an *immutable promoter*.

Generic rules: To process complex symbols, we use high-level generic rules, which are *instantiated* using *free variable* matching [1]. A generic rule is identified by an extended version of the classical rewriting mode, in fact, a combined *instantiation.rewriting* mode, where (1) the *instantiation* mode is one of {`min`, `max`, `dyn`} and (2) the *rewriting* mode is one of {`min`, `max`}.

The instantiation mode indicates how many instance rules are conceptually generated: (a) the mode `min` indicates that the generic rules is nondeterministically instantiated only *once*, if possible; (b) the mode `max` indicates that the generic rule is instantiated as *many* times as possible, without *superfluous* instances (i.e. without duplicates or instances which are not applicable); (c) the newly proposed mode `dyn` indicates a dynamic instantiation mode, which will be described later. The rewriting mode indicates how each instantiated rule is applied (as in the classical framework).

As an example, consider a system where cell σ_7 contains multiset $f_2 {f_3}^2 v$, and the generic rule ρ_α, where $\alpha \in \{\texttt{min.min}, \texttt{min.max}, \texttt{max.min}, \texttt{max.max}\}$ and i and j are free variables:

$$(\rho_\alpha)\ S_{20}\ f_j\ \rightarrow_\alpha\ S_{20}\ (b_i)\updownarrow_j\ \mid v\ \iota_i$$

1. $\rho_{\texttt{min.min}}$ nondeterministically generates *one* of the following rule instances:

$$S_{20}\ f_2\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_2$$
$$S_{20}\ f_3\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_3$$

2. $\rho_{\texttt{min.max}}$ nondeterministically generates *one* of the following rule instances:

$$S_{20}\ f_2\ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_2$$
$$S_{20}\ f_3\ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_3$$

3. $\rho_{\texttt{max.min}}$ generates *both* following rule instances:

$$S_{20}\ f_2\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_2$$

$$S_{20}\ f_3\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_3$$

4. $\rho_{\texttt{max.max}}$ generates *both* following rule instances:

$$S_{20}\ f_2\ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_2$$
$$S_{20}\ f_3\ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_3$$

In a *matrix* organized ruleset, if a generic rule using the `max` instantiation mode generates more than one simple rule, then all generated rules take the generic rule's place in the vector, in some *nondeterministic* order.

Like `max`, `dyn` instantiation mode has the potential to generate any number of rules (depending on the actual cell contents). Like `min`, `dyn` starts by generating one possible instance. However, after the generated rule is applied, `dyn` repeats the generation process, until either no new rules can be generated or a specified bound has been reached (by default, we use the cell's *degree*).

As an example, consider a cell containing the following list of complex symbols: $m(c^{i_0})$, $a_1(c^{i_1})$, $a_2(c^{i_2})$, ..., $a_n(c^{i_n})$, representing the values i_0, i_1, i_2, ..., i_n, respectively (where $n \geq 0$). The following generic rule, μ, determines the minimum over this sequence of values, in one single step:

$$(\mu)\ S_0\ m(XY)\ \rightarrow_{\texttt{dyn.min}}\ S_0\ m(X) \mid a_j(X)$$

Assume the particular scenario when $n = 3$, $i_0 = 4$, $i_1 = 7$, $i_2 = 2$, $i_3 = 3$, i.e. our cell contains $m(c^4)$, $a_1(c^7)$, $a_2(c^2)$, $a_3(c^3)$. First, μ instantiates *one* of the following rules, μ' or μ'':

$$(\mu')\ S_0\ m(c^2c^2)\ \rightarrow_{\texttt{min}}\ S_0\ m(c^2) \mid a_2(c^2)$$
$$(\mu'')\ S_0\ m(c^3c)\ \rightarrow_{\texttt{min}}\ S_0\ m(c^3) \mid a_3(c^3)$$

If generated, rule μ' transforms $m(c^4)$ into $m(c^2)$, which indicates the required minimum, $2 = \min(4, 7, 2, 3)$. Otherwise, rule μ'' transforms $m(c^4)$ into $m(c^3)$ and then the `dyn` mode instantiates another rule, μ''', which determines the required minimum:

$$(\mu''')\ S_0\ m(c^2c)\ \rightarrow_{\texttt{min}}\ S_0\ m(c^2) \mid a_2(c^2)$$

The *matrix* organised rulesets and the `dyn` instantiation have been specifically designed to *level the playing field* between P systems and the usual frameworks used in distributed algorithms. Typically, distributed algorithms steps only count messaging rounds, ignoring local computations; therefore, a node in a distributed algorithm can determine the minimum over an arbitrary long local sequence in one single step.

The instantiation of generic rules is only *conceptual*: it explains their high-level semantics by mapping it to a simpler lower-level semantics. Moreover, this instantiation is also *ephemeral*: the generated lower-level rules are not supposed to exist past the end of the step. An actual P system implementation does not need to effectively use rule instantiation, as long as it can support the same high-level semantics by other means.

5 P System Specification

In this section, we present a directly executable P system specification of Algorithm D, having the *same distributed runtime complexity*. We omit Algorithm C, which is contained in its extension, Algorithm D.

The input digraph is given by the P system structure itself and the system is fully distributed, i.e. there is no central node and only local messaging channels (between structural neighbours) are allowed. Moreover, we consider that cells start without any kind of network topology awareness: cells do not know the identities of their children, not even their numbers.

The P specification has a challenging task: to fully formalize the informal description given by the high-level pseudocodes, completing all important details ignored by these, all this without increasing the time complexity. The specification needs to indicate how to build local digraph neighbourhood awareness, how to build and navigate over virtual residual digraphs, how to transform augmenting paths into edge-disjoint paths, how to discard "dead" cells, how to manage concurrent notification processes.

In particular, our pseudocodes use structural and virtual arcs between cells: in the corresponding P system specification, parent and child cells record their corresponding arc end-points, building a simple form of distributed routing tables (such as used in networking).

P Specification 1: Algorithm D

Input: All cells start with the same set of rules and without any topological awareness (they do not even know their local neighbours or even their numbers). All cells start in the same initial state, S_0. Initially, each cell, σ_i, contains an immutable cell ID symbol, ι_i. Additionally, the source cell, σ_s, and the target cell, σ_t, are decorated with symbols, s and t, respectively.

Output: All cells end in the *same final state*, S_{60}. On completion, all cells are *empty*, with exceptions: (1) The source cell, σ_s, and the target cell, σ_t, are still decorated with symbols, s and t, respectively; (2) The cells on edge-disjoint paths contain path link symbols, for predecessors, d'_j, and successors, d''_k.

Table 1 shows the initial and final cells' configurations for the P system based on the digraph illustrated in **Figure 1**.

Table 1: Initial and final configurations of P Specification 1, for Figure 1.

Cell	σ_0	σ_1	σ_2	σ_3
Initial	$S_0\ \iota_0\ s$	$S_0\ \iota_1$	$S_0\ \iota_2$	$S_0\ \iota_3$
Final	$S_{60}\ \iota_0\ s\ d''_1\ d''_2\ d''_3$	$S_{60}\ \iota_1\ d'_0\ d''_4$	$S_{60}\ \iota_2\ d'_0\ d''_6$	$S_{60}\ \iota_3\ d'_0\ d''_5$
Cell	σ_4	σ_5	σ_6	σ_7
Initial	$S_0\ \iota_4$	$S_0\ \iota_5$	$S_0\ \iota_6$	$S_0\ \iota_7\ t$
Final	$S_{60}\ \iota_4\ d'_1\ d''_7$	$S_{60}\ \iota_5\ d'_3\ d''_7$	$S_{60}\ \iota_6\ d'_2\ d''_7$	$S_{60}\ \iota_7\ t\ d'_4\ d'_5\ d'_6$

The matrix R of P Specification 1 consists of fifteen vectors, informally presented in five groups, according to their functionality and applicability. Each vector implements an independent function, performed in one step. Symbols i, j and k are free variables related

to cell IDs, symbols X and Y are free variables which match multisets; conventionally, we use i, j and k as subscripts and X and Y as arguments.

0. Shared start (S_0–S_2)

0.1.

1. $S_0\ n\ \rightarrow_{\texttt{min.min}}\ S_1\ (n)\updownarrow_\forall\ (n''_i)\uparrow_\forall\ (n'_i)\downarrow_\forall\ \mid \iota_i$

2. $S_0\ \rightarrow_{\texttt{min}}\ S_0\ n \mid s$

0.2.

1. $S_1\ \rightarrow_{\texttt{min}}\ S_2$

0.3.

1. $S_2\ n\ \rightarrow_{\texttt{max}}\ S_3$

2. $S_2\ \rightarrow_{\texttt{min}}\ S_3$

1. Initial differentiation (S_3): cf. Lines 3.4-7
1.1.

1. $S_3\ \rightarrow_{\texttt{min}}\ S_{10}\ f\ r_i(c)\ (w_i\ v_i)\updownarrow_\forall\ \mid \iota_i\ s$

2. $S_3\ \rightarrow_{\texttt{min}}\ S_{30} \mid t$

3. $S_3\ \rightarrow_{\texttt{min}}\ S_{20}$

2. Source cell (S_{10}): cf. Lines 3.8-14, 7.15, 3.20-22
2.1.

1. $S_{10}\ f\ \rightarrow_{\texttt{min.min}}$
   $S_{10}\ s''_k\ (f_i\ r_i(Xc))\downarrow_k\ \mid n''_k\ h(X)\ \iota_i\ \neg\ w_k\ v_k$

2. $S_{10}\ a\ s''_k\ n''_k\ \rightarrow_{\texttt{min.min}}\ S_{40}\ p\ d''_k\ (p)\updownarrow$

3. $S_{10}\ a\ \rightarrow_{\texttt{max}}\ S_{40}$

4. $S_{10}\ b_k\ s''_k\ n''_k\ \rightarrow_{\texttt{min.min}}\ S_{40}\ q\ (q)\updownarrow$

5. $S_{10}\ b_k\ \rightarrow_{\texttt{max.max}}\ S_{40}$

6. $S_{10}\ f\ \rightarrow_{\texttt{min.min}}\ S_{50}\ (g)\updownarrow$

3. Intermediate cells (S_{20})

3.1. Finalisation: cf. Line 3.22

1. $S_{20}\ g\ \rightarrow_{\texttt{min.min}}\ S_{50}\ (g)\updownarrow$

3.2. Frontier: cf. Lines 4.5-26

1. $S_{20}\ \rightarrow_{\texttt{min.min}}\ S_{20}\ r_i(X)(r_i(X))\updownarrow_\forall\ |f_j h(X)\iota_i$

2. $S_{20}\ f_j\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v\ s'_j\ (v_i)\updownarrow_\forall\ f \mid \iota_i$

3. $S_{20}\ r_k(X)\ r'_k(Y)\ \rightarrow_{\texttt{min.min}}\ S_{20}\ r_k(Y) \mid b_k$

4. $S_{20} \rightarrow_{\texttt{min.min}} S_{20}(x)\updownarrow_k |h(X)r_k(XY)b_k\iota_i\neg w_k$

5. $S_{20}\ b_k\ s''_k\ \rightarrow_{\texttt{min.min}}\ S_{20}\ f\ z''_k$

6. $S_{20}\ b_k\ \rightarrow_{\texttt{max.max}}\ S_{20}$

7. $S_{20}\ \rightarrow_{\texttt{min.min}}\ S_{20}\ n(X) \mid h(X)\ f\ \iota_i$

8. $S_{20}\ n(XY) \rightarrow_{\texttt{dyn.min}} S_{20}\ n(X)|r_j(X)n''_j v_j f\iota_i$

9. $S_{20}\ n(XY) \rightarrow_{\texttt{dyn.min}} S_{20}\ n(X)|r_j(X)d'_j v_j f\iota_i$

10. $S_{20}\ n(X)\ r_i(XY)\ \rightarrow_{\texttt{min.min}}$
    $S_{20}\ r_i(X)\ (r'_i(X))\updownarrow_\forall\ \mid \iota_i$

11. $S_{20}\ f\ \rightarrow_{\texttt{min.min}}$
    $S_{20}\ v_k\ s''_k(f_i h(Xc))\downarrow_k |\iota_i\ n''_k\ h(X)\neg v_k\ d'_k\ d''_k$

12. $S_{20}\ f\ \rightarrow_{\texttt{min.min}}$
    $S_{20}\ v_k\ s''_k(f_i\ h(Xc))\uparrow_k |\iota_i\ d'_k\ h(X)\ \neg\ v_k$

13. $S_{20}\ f\ s'_j\ \rightarrow_{\texttt{min.min}}\ S_{20}\ (b_i)\updownarrow_j\ \mid \iota_i$

14. $S_{20}\ n(X)\ \rightarrow_{\texttt{min.min}}\ S_{20}$

3.3. Path confirmation: cf. Lines 3.16-19

1. $S_{20}\ a\ s'_j\ s''_k\ \rightarrow_{\texttt{min.min}}\ S_{20}\ d'_j\ d''_k\ (a)\updownarrow_j$

2. $S_{20}\ a\ \rightarrow_{\texttt{max}}\ S_{20}$

3. $S_{20}\ d''_k\ d'_k\ \rightarrow_{\texttt{min.min}}\ S_{20}$

3.4. End-of-round resets: cf. Lines 7.15, 3.20

1. $S_{20}\ \rightarrow_{\texttt{min}}\ S_{21}\ (q)\updownarrow_\forall\ \mid q$

2. $S_{20}\ \rightarrow_{\texttt{min}}\ S_{21}\ w \mid q\ v\ \neg\ w$

3. $S_{20}\ \rightarrow_{\texttt{max.min}}\ S_{21}\ w_k \mid q\ v_k\ \neg\ w_k$

4. $S_{20}\ z''_k\ \rightarrow_{\texttt{min}}\ S_{21} \mid q$

5. $S_{20}\ \rightarrow_{\texttt{min}}\ S_{21}\ (p)\updownarrow_\forall\ \mid p$

6. $S_{20}\ v\ \rightarrow_{\texttt{min}}\ S_{21} \mid p\ \neg\ w$

7. $S_{20}\ v_k\ \rightarrow_{\texttt{max.min}}\ S_{21} \mid p\ \neg\ w_k$

3.5. Transit to the end of a search round

1. $S_{21}\ \rightarrow_{\texttt{min}}\ S_{40}$

3.6. Update: cf. Lines 6.4-15

1. $S_{20}\ r_j(X)\ \rightarrow_{\texttt{max.max}}\ S_{20} \mid r_j(X)$

2. $S_{20}\ r'_j(X)\ \rightarrow_{\texttt{max.max}}\ S_{20} \mid r'_j(X)$

3. $S_{20}\ r_j(X)\ r'_j(Y)\ \rightarrow_{\texttt{min.min}}\ S_{20}\ r_j(Y)$

4. $S_{20}\ \rightarrow_{\texttt{min.min}}\ S_{20}\ n(X) \mid h(X)\ u_k\ \iota_i$

5. $S_{20}n(XY) \rightarrow_{\texttt{dyn.min}} S_{20}n(X)|r_j(X)n''_j v_j u_k \iota_i$

6. $S_{20}n(XY) \rightarrow_{\texttt{dyn.min}} S_{20}n(X)|r_j(X)d'_j v_j u_k \iota_i$

7. $S_{20}\ n(XY)\ r_i(X)\ \rightarrow_{\texttt{min.min}}$
   $S_{20}\ r_i(XY)\ (r'_i(XY)\ u_i)\updownarrow_\forall |u_k\ n''_k\ v_k\ \iota_i\neg w$

8. $S_{20}\ n(XY)\ r_i(X)\ \rightarrow_{\texttt{min.min}}$
   $S_{20}\ r_i(XY)\ (r'_i(XY)\ u_i)\updownarrow_\forall |u_k\ d'_k\ v_k\ \iota_i\neg w$

9. $S_{20}\ n(X)\ \rightarrow_{\texttt{min.min}}\ S_{20}$

10. $S_{20}\ u_k\ \rightarrow_{\texttt{min.min}}$
$S_{20}\ (x)\updownarrow_k\ \mid h(X)\ r_k(XY)\ \iota_i\ \neg\ w_k\ w$

11. $S_{20}\ u_k\ \rightarrow_{\texttt{max.max}}\ S_{20}$

3.7. Discard: cf. Lines 5.3-15

1. $S_{20}\ z''_k\ \rightarrow_{\texttt{max.min}}\ S_{20}\ (x)\updownarrow_k\ \mid x\ \iota_i\ \neg\ w_k$

2. $S_{20}\ x\ \rightarrow_{\texttt{min.min}}$
$S_{20}\ w\ r_i(\infty)\ (w_i\ r'_i(\infty)\ u_i)\updownarrow_\forall\ \mid \iota_i\ \neg\ w$

3. $S_{20}\ z''_k\ \rightarrow_{\texttt{max.max}}\ S_{20} \mid w$

4. $S_{20}\ s'_j\ s''_k\ \rightarrow_{\texttt{min.min}}\ S_{20}\ (b_i)\updownarrow_j\ \mid r_i(X)\ w\ \iota_i$

4. Target cell (S_{30}): cf. Line 4.4

4.1.

1. $S_{30}\ g\ \rightarrow_{\texttt{min.min}}\ S_{40}\ (g)\updownarrow$

2. $S_{30}\ f_j\ \rightarrow_{\texttt{min.min}}\ S_{30}\ d'_j\ (a)\downarrow_j$

3. $S_{30}\ \rightarrow_{\texttt{min}}\ S_{40} \mid q$

4. $S_{30}\ \rightarrow_{\texttt{min}}\ S_{40} \mid p$

5. All cells (S_{40}, S_{50})

5.1. End of each search round

1. $S_{40}\ v_k\ \rightarrow_{\texttt{max.max}}\ S_{40} \mid s$

2. $S_{40}\ v_k\ \rightarrow_{\texttt{max.max}}\ S_{40} \mid t$

3. $S_{40}\ u_j\ \rightarrow_{\texttt{max.max}}\ S_{40}$

4. $S_{40}\ c_l\ \rightarrow_{\texttt{max.max}}\ S_{40}$

5. $S_{40}\ r_j(X)\ \rightarrow_{\texttt{max.max}}\ S_{40}$

6. $S_{40}\ r'_j(X)\ \rightarrow_{\texttt{max.max}}\ S_{40}$

7. $S_{40}\ a\ \rightarrow_{\texttt{max}}\ S_{40}$

8. $S_{40}\ q\ \rightarrow_{\texttt{max}}\ S_{40}$

9. $S_{40}\ p\ \rightarrow_{\texttt{max}}\ S_{40}$

10. $S_{40}\ \rightarrow_{\texttt{min}}\ S_3$

5.2. End of the algorithm

1. $S_{50}\ g\ \rightarrow_{\texttt{max}}\ S_{50}$

2. $S_{50}\ n'_j\ \rightarrow_{\texttt{max.min}}\ S_{50}$

3. $S_{50}\ n''_k\ \rightarrow_{\texttt{max.min}}\ S_{50}$

4. $S_{50}\ w_k\ \rightarrow_{\texttt{max.min}}\ S_{50}$

5. $S_{50}\ v_k\ \rightarrow_{\texttt{max.min}}\ S_{50}$

6. $S_{50}\ z''_k\ \rightarrow_{\texttt{max.min}}\ S_{50}$

7. $S_{50}\ w\ \rightarrow_{\texttt{max}}\ S_{50}$

8. $S_{50}\ v\ \rightarrow_{\texttt{max}}\ S_{50}$

9. $S_{50}\ \rightarrow_{\texttt{min}}\ S_{60}$

Cell σ_i uses the following symbols to record its relationships with its neighbouring cells, σ_j and σ_k: n'_j indicates a structural parent; n''_k indicates a structural child; d'_j indicates an edge-disjoint path predecessor (dp-predecessor); d''_k indicates an edge-disjoint path successor (dp-successor); s'_j indicates a current sp-predecessor; s''_k indicates a current sp-successor; z''_k indicates a st-successor; $r_j(c^m)$ records σ_j's reach-number, m (note that here j may indicate the current cell, i, or one of its neighbours).

Additionally, cell σ_i uses the following symbols to record its state: $h(c^m)$ records its depth, m; $n(c^m)$ is used to evaluate the minimum over its own depth and the reach-numbers of its temporarily visited structural children; v indicates that it is temporarily visited; w indicates that it is permanently visited; f indicates that it is the frontier cell.

Cell σ_i sends out messages consisting of the following symbols: f_i is the forward token; b_i is the backtrack token; v_i is the visited notification; w_i is the permanently visited notification; $r'_i(c^m)$ is its updated reach-number, m; x is the discarding notification; a is the path confirmation; q is the after-success reset; p is the after-failure reset; g is the finalise token.

Here we explain a small snippet, vector 3.2, which contains several critical rules for an intermediate cell, σ_i.

Note that (as indicated above), we use two distinct symbols to represent the *visit token*: a *forward token* and *backtrack token*. Each token carries the sender ID: f_i is the

forward token sent by σ_i and b_i is the backtrack token sent by σ_i.

Rules 3.2.1–2 process an incoming *forward token*, f_j, from cell σ_j. If it is unvisited, $\neg\ v$, then cell σ_i (a) initialises its reach-number, $r_i(X)$, as the received depth, $h(X)$; (b) becomes visited, v; (c) records σ_j as its current sp-predecessor, s'_j; (d) broadcasts its visited notification, v_i, to all its neighbours, $\updownarrow_\forall$; and (e) becomes the *search frontier*, f.

Rules 3.2.4–7 process an incoming *backtrack token*, b_k, from cell σ_k. Cell σ_i (a) updates its record for σ_k's reach-number, $r_k(X)$; (b) sends a discarding notification to σ_k, if σ_k's reach-number is greater than σ_i's depth; (c) transforms its current sp-successor, s''_k, into a st-successor, z''_k; and (d) becomes the *search frontier*, f.

Rules 3.2.8–14 specify the behaviour of cell σ_i, after it becomes the *search frontier*, f. Rules 3.2.8–10 compute σ_i's reach-number as the *minimum* of its depth, $h(X)$, and the reach-numbers of its temporarily visited residual digraph children, i.e. temporarily visited structural children (n''_k and v_k) or temporarily visited dp-predecessors (d''_k and v_k). Rule 3.2.11 updates and broadcasts σ_i's reach-number, $r'_i(X)$, if this value decreases.

According to rule 3.2.12, if σ_k is an *unvisited structural child*, $n''_k\ \neg\ v_k$, that is not on an existing disjoint path, $\neg\ d'_k\ d''_k$, then σ_i (a) records σ_k as visited, v_k; (b) records σ_k as its current sp-successor, s''_k; and (c) sends its *forward token*, f_i, with an incremented depth, $h(Xc)$, to σ_k, over an outgoing structural arc, $\downarrow_k$ (i.e. over a direct arc).

If the conditions of rule 3.2.12 are not met (rules are applied in the weak priority order), then rule 3.2.13 is considered. According to rule 3.2.14, if σ_k is an *unvisited dp-predecessor*, $\neg\ v_k$ and d'_k, then σ_i (a) records σ_k as visited, v_k; (b) records σ_k as its current sp-successor, s''_k; and (c) sends its *forward token*, f_i, to σ_k, with an incremented depth, $h(Xc)$, over an incoming structural arc, $\uparrow_k$ (over a reverse arc).

If the conditions of rules 3.2.12–13 are not met, then rule 3.2.14 is considered. According to rule 3.2.14, if σ_j is the *current sp-predecessor*, s'_j, then σ_i (a) removes s'_j, i.e. the existing record of σ_j as its current sp-predecessor; and (b) sends its *backtrack token*, b_i, to σ_j, over an outgoing or incoming structural arc, $\updownarrow_j$ (over a direct or reverse arc).

6 Runtime Performance

Consider a digraph with n cells and m arcs, where d is the outdegree of the source cell and f is the maximum number of edge-disjoint paths in a given scenario. In Algorithms B, C, C* and D, the source cell starts d search rounds. In each round, using a Cidon-type DFS, visited cells notify their neighbours, so the search does not revisit cells which were visited in the same round and thus completes in at most n steps. As earlier mentioned, all other housekeeping operations are performed in parallel with the main search, thus Algorithms B, C, C* and D all run in $O(nd)$ steps. In fact, because they discard all cells visited in *failed* rounds (which do not find augmenting paths), Algorithms B and D run in $O(nf)$ steps. We conjecture that a similar upperbound can also be found for C and C*.

Proposition 2. Algorithms C and C* run in $O(nd)$ steps; Algorithms B and D run in $O(nf)$ steps.

Table 2 compares the asymptotic complexity of our new Algorithms C, C* and D

Table 2: Asymptotic worst-case complexity of distributed DFS-based algorithms.

Algorithm	**Runtime Complexity**
Algorithm A (Ford-Fulkerson/DFS [5])	$O(mf)$ steps
Algorithm A* (Dinneen et al. [3])	$O(mf)$ steps
Algorithm B (our previous improvement [10])	$O(nf)$ steps
Algorithm C (here)	$O(nd)$ steps (?)
Algorithm C* (here)	$O(nd)$ steps (?)
Algorithm D (here)	$O(nf)$ steps

against our previous Algorithm B and the two other previous DFS-based algorithms used in this paper.

However, this theoretical estimation does not fully account for the detection and discarding of "dead" (permanently visited) cells. Therefore, using their executable P specifications, we experimentally compare Algorithms A, B, C, C* and D, on thirty digraphs with 100 cells and 300 arcs, generated using NetworkX [7]. Algorithm C* is a restricted version of Algorithm C, which, using our novel idea, only discards "dead" cells detected during *successful* rounds, intentionally refraining from discarding any "dead" cells detected during *failed* rounds. This way, Algorithm C* is the opposite of our previous Algorithm B, which, using a different idea, only discards "dead" cells detected during *failed* rounds.

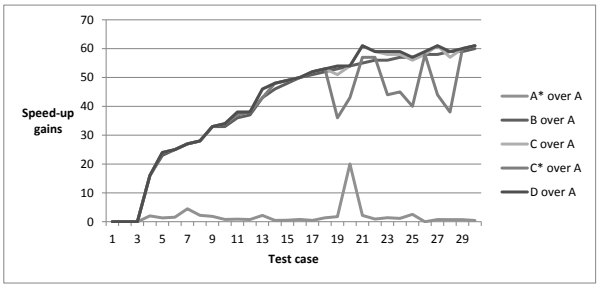

Figure 3: Speed-up gains of Algorithms A*, B, C, C* and D over A for thirty test cases.

Figure 3 shows the speed-up gains of Algorithms B, C, C* and D over A for our thirty test cases. On average, (1) Algorithm B is 41.0% faster than Algorithm A; (2) Algorithm C is 41.8% faster than Algorithm A; (3) Algorithm C* is 38.0% faster than Algorithm A; (4) Algorithm D is 42.1% faster than Algorithm A.

Analysing results (1–3), we found that the "dead" cells *detected* by Algorithms C and C* do cover all "dead" cells detected by Algorithm B, and even a few more. However, not all detected "dead" cells can be effectively *discarded* in "real-time", unless we allow these two algorithms to run longer (which we do not want). Thus, we can find (1) scenarios, such as shown in **Figure 2**, where Algorithms C and C* (and, of course, Algorithm D) outperform Algorithm B, and (2) scenarios, such as shown in **Figure 4**, where Algorithm B runs faster than Algorithms C and C* (but not than D).

In **Figure 4**, Algorithm C does detect all "dead" cells detected by Algorithm B, but, because of pruning propagating delays, does not effectively discard them in "real-time"; briefly, it does not show the same runtime performance.

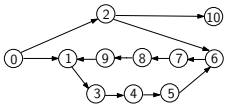

Figure 4: An example, in which Algorithm B performs better than Algorithm C.

For Algorithm C, when round #1 search path $\tau = \sigma_0.\sigma_1.\sigma_3.\sigma_4.\sigma_5.\sigma_6.\sigma_7.\sigma_8.\sigma_9$, backtracks to the source cell, σ_0, cells $\sigma_i, i \in \{1\} \cup [3, 9]$ can be discarded, because their reach-numbers are greater than $\sigma_0.\texttt{depth} = 0$. Cell σ_0 triggers a discarding notification, which follows the *same path* as the backtracked search τ. In round #2, the *new* search path τ', $\tau' = \sigma_0.\sigma_2$ visits σ_6 *before* this cell receives its due discarding notification and then continues to σ_7 and further. Later, cell σ_6 receives its due discarding notification (started in round #1), discards itself and sends an overdue backtrack token to σ_2, which starts looking for other directions to continue path τ'. However, several steps have been lost exploring "dead" nodes (which were not aware of this). Finally, after this mentioned delay, the search path τ' reaches σ_{10}, $\tau' = \sigma_0.\sigma_2.\sigma_{10}$, and becomes a new augmenting path.

For Algorithm B, the round #1 search path, τ, follows the same route as in Algorithm C. When τ backtracks to σ_0, Algorithm B initiates an after-failure reset, which is propagated as a *broadcast*, travelling on *shortest paths*, reaching σ_6 on path $\sigma_0.\sigma_2.\sigma_6$. When the immediately following round #2 search path τ' reaches σ_2, $\tau' = \sigma_0.\sigma_2$, it avoids σ_6, which is already discarded. In the next step, the search path τ' reaches σ_{10}, $\tau' = \sigma_0.\sigma_2.\sigma_{10}$, and becomes a new augmenting path, faster than in Algorithm C.

In this example, due to its pruning propagating delay, Algorithm C shows worse performance than Algorithm B: in their executable P specification, Algorithm C requires 46 steps, while Algorithm B takes only 41 steps.

In **Figure 2**, Algorithm C outperforms Algorithm B. For Algorithm C, when the round #1 search path, τ, backtracks to σ_3 (see (d)), σ_6 can be discarded and is sent a discarding notification. Later, when τ backtracks to σ_2 (see (e)), σ_3, σ_4 and σ_5 can be discarded and are sent discarding notifications. All these discarding notifications reach their targets before the start of the next round. Thus, a round #2 search path, τ', will not (needlessly) probe σ_4 and its descendants.

In contrast, Algorithm B, which uses a different idea, cannot detect "dead" cells during successful rounds. Its round #1 search path, τ, follows the same route as in Algorithm C; however, without triggering any discarding notification. Therefore, a round #2 search path, τ', will needlessly visit again cells σ_4, σ_5, σ_3 and σ_6. In their executable P specification, Algorithm C takes 30 steps, while Algorithm B takes 43 steps.

Acknowledgment

The author wishes to thank the assistance received via the University of Auckland FRDF grant 9843/3626216.

7 Conclusions

We presented two new distributed DFS-based algorithms, Algorithms C and D, for solving the edge-disjoint path problem in digraphs. Using a novel idea, Algorithm C discards "dead" cells detected during both successful and failed search branches. By combining Algorithm C and our previous Algorithm B [10], which discards "dead" cells detected during failed rounds, Algorithm D discards all "dead" cells that are detected by both B and C.

We first described our distributed algorithms using informal high-level pseudocodes and then we provided an equivalent directly executable formal P specification. Our P systems use high-level generic rules organised in a newly proposed matrix-like structure and with a new `dyn` instantiation mode. The resulting P systems have a reasonably fixed-sized ruleset, i.e. the number of rules does not depend on the number of cells, and achieve the same runtime complexity as the corresponding distributed algorithms.

Experimentally, on a series of random digraphs, all our algorithms seem to show very significant speed-up over the classical Algorithm A and its improved version, Algorithm A*. Interestingly, despite using a different idea, our new algorithms seem to have a similar performance with our previous Algorithm B; in fact, on purely *random* digraphs, Algorithms C and D seem to be marginally faster than Algorithm B.

On the other side, one can construct many sample scenarios where Algorithms C and D vastly outperform Algorithm B and also many sample scenarios where Algorithm B outperforms Algorithm C (but not Algorithm D).

Several interesting questions remain open. Can these results be extrapolated to digraphs with different characteristics, such as size, average node degree, node degree distribution? Will these results remain valid for symmetric digraphs, i.e., undirected graphs? Can we find improved versions of these algorithms for solving the undirected graph problem? How relevant are these algorithms and results for real-life networks, such as transportation networks or other networks which show some kind of clustering? Are there well defined practical (non-random) scenarios where one could recommend one of the algorithm over another? Can we apply similar optimisations to BFS-based algorithms for solving the edge-disjoint paths problem? What are practical strengths and limits of P systems based on our matrix structured generic rules?

References

[1] Bălănescu, T., Nicolescu, R., Wu, H.: Asynchronous P systems. International Journal of Natural Computing Research 2(2), 1–18 (2011)

[2] Cidon, I.: Yet another distributed depth-first-search algorithm. Inf. Process. Lett. 26, 301–305 (January 1988)

[3] 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)

[4] Dinneen, M.J., Kim, Y.B., Nicolescu, R.: A faster P solution for the Byzantine agreement problem. In: Gheorghe, M., Hinze, T., Păun, G. (eds.) Conference on Membrane Computing. Lecture Notes in Computer Science, vol. 6501, pp. 175–197. Springer-Verlag, Berlin Heidelberg (2010)

[5] Ford, L.R., Jr., Fulkerson, D.R.: Maximal flow through a network. Canadian Journal of Mathematics 8, 399–404 (1956)

[6] Freund, R., Păun, G.: A variant of team cooperation in grammar systems. Journal of Universal Computer Science 1(2), 105–130 (1995)

[7] Hagberg, A.A., Schult, D.A., Swart, P.J.: Exploring Network Structure, Dynamics, and Function using NetworkX. In: Varoquaux, G., Vaught, T., Millman, J. (eds.) 7th Python in Science Conference (SciPy). pp. 11–15 (2008)

[8] Karp, R.M.: Reducibility Among Combinatorial Problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press (1972)

[9] Nicolescu, R.: Parallel and distributed algorithms in P systems. Lecture Notes in Computer Science, vol. 7184, pp. 33–42. Springer-Verlag (2012)

[10] Nicolescu, R., Wu, H.: New solutions for disjoint paths in P systems. Natural Computing pp. 1–15 (2012), 10.1007/s11047-012-9342-9

[11] Păun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000)

[12] Păun, G., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)

[13] Tel, G.: Introduction to Distributed Algorithms. Cambridge University Press (2000)