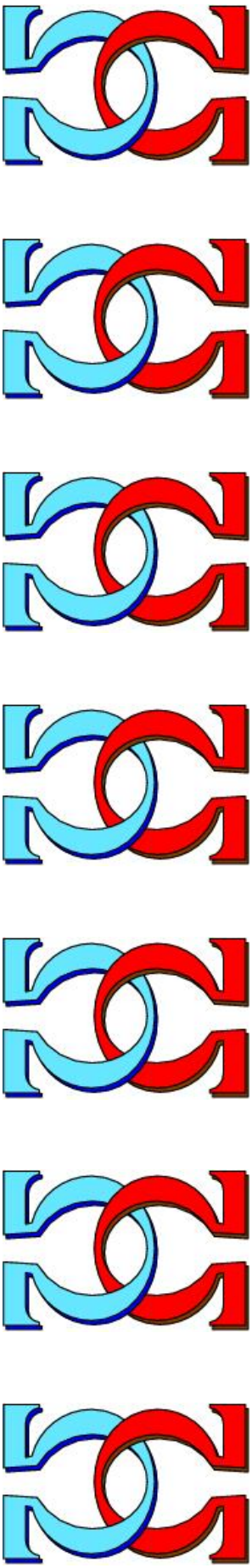


CDMTCS
Research
Report
Series

New Solutions for Disjoint Paths in P Systems

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

CDMTCS-417
March 2012

Centre for Discrete Mathematics and
Theoretical Computer Science

New Solutions for Disjoint Paths in P Systems

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

Abstract

We propose faster algorithms to identify maximum cardinality sets of edge- and node-disjoint paths, between a source cell and a target cell in P systems. Previously, Dinneen et al. presented two algorithms, based on classical depth-first search (DFS), which run in $O(md)$ P steps; where m is the number of edges and d is the outdegree of the source node. Combining Cidon's distributed DFS and our new result, Theorem 6, we propose two improved DFS-based algorithms, which run in $O(nd)$ P steps; where n is the number of nodes. We also propose two improved algorithms, based on breadth-first search (BFS), with $O(nf)$ runtime complexity, where f is the number of disjoint paths. All our algorithms are fixed-size, i.e. the number of P rules does not depend on the number of cells in the underlying P system. Empirically, for randomly generated digraphs of various sizes, our DFS-based edge-disjoint algorithm is 39% faster than Dinneen et al.'s edge-disjoint algorithm and our BFS-based edge-disjoint algorithm is 80% faster, in terms of P steps. Our improvements can be considered for any distributed DFS implementation (i.e. not only for P systems).

Keywords: breadth-first search, Cidon's depth-first search, depth-first search, edge-disjoint paths, node-disjoint paths, network flow, P systems

1 Introduction

A P system is a parallel and distributed computational model inspired by the structure and interactions of cell membranes, introduced by Păun [15, 16]. All cells evolve synchronously by applying rules in a non-deterministic and (potentially maximally) parallel manner.

Alternative paths between two nodes are important in many fields. They are fundamental in biological remodelling, e.g., of nervous or vascular systems [7]. Multipath routing provides effective bandwidth in networks. Disjoint paths are sought in streaming multi-core applications to avoid sharing communication links between processors [17]. The maximum matching problem in a bipartite graph can be transformed to the disjoint

paths problem [3]. For non-complete graphs, Byzantine agreement requires at least $2k+1$ node-disjoint paths, between each pair of nodes to ensure that a distributed consensus can occur, with up to k failures [12].

In a native P system solution for the disjoint path problem, the input graph should be given by the P system structure itself, more specifically, by its topology. The system is fully distributed, i.e. there is no central node and only local channels (between structural neighbours) are allowed.

Dinneen et al. proposed the first P solutions [4], here called DKN-DFS-Edge and DKN-DFS-Node, as distributed versions of the Ford-Fulkerson algorithm [8], based on the classical DFS [18]. However, these solutions are not optimal: (i) they use the classical DFS, which does not leverage the distributed features of P systems and (ii) subsequent rounds keep revisiting all cells which were visited in previous failed rounds, although these (as we prove) can never lead to any successful search path.

We propose two faster DFS solutions: P Algorithm 1—NW-DFS-Edge (see Section 5.1) and P Algorithm 2—NW-DFS-Node (see Section 5.2). First, we use the basic idea of Cidon's distributed DFS algorithm [2], which avoids forwarding the search token to cells previously visited in the same round. Secondly, we present and prove a new result, Theorem 6 (see Section 5), which further speeds up our algorithms, by discarding cells visited during previous failed rounds, i.e. cells which can remain *permanently* visited. Finally, we propose two improved versions of our earlier BFS-based algorithms [14]: P Algorithm 3—NW-BFS-Edge and (see Section 6.1) and P Algorithm 4—NW-BFS-Node (see Section 6.2). These algorithms represent distributed versions of the Edmonds-Karp algorithm [6], which leverage the parallel potential of P systems, to concurrently search as many paths as possible. All our algorithms work for digraphs.

2 Preliminaries

Essentially, a P system is specified by its membrane structure, symbols and rules. The membrane structure is a *directed graph* (*digraph*) or one of its subclasses, such as a *directed acyclic graph* (*DAG*) or, occasionally, a (rooted) *tree*, or a more complex structure, such as a *hypergraph* or a *multigraph*; (undirected) *graph* structures can be emulated by symmetric digraphs. In this paper, our membrane structure is a *digraph*.

Digraph arcs define structural (*parent*, *child*) relationships and, in diagrams, are represented as arrows from parents to children. Arcs function as *duplex* channels, i.e. messages can travel in both directions: from parents to children and from children to parents.

Each arc has *two* labels, (x, y), which can be used for directing messages to a specific destination: (i) one label, x, on the parent side, referring to the child, and (ii) another label, y, on the child side, referring to the parent. This is an extension of the usual graph convention, where an arc has just one label. Labels can be explicitly indicated; otherwise, we assume a default labelling convention. An unlabelled arc, (σ_i, σ_j), is implicitly labelled with the indices of its two adjacent cells, in reverse order: (j, i).

In this paper, we use an extended version of *simple P modules*, as defined in [5].

Definition 1 (Simple P module). A simple P module with duplex channels is a system $\Pi = (O, V, E)$, where O is a finite non-empty alphabet of symbols; V is a finite set of

cells and E is a set of labelled digraph arcs on V, representing a structural *parent-child* relationship between cells, with *duplex* communication capabilities.

Each cell, $\sigma_i \in V$, has the initial configuration $(Q_i, S_{i0}, w_{i0}, R_i)$, and the current configuration (Q_i, S_i, w_i, R_i), where Q_i is a finite set of states; $S_{i0} \in Q_i$ is the initial state; $S_i \in Q_i$ is the current state; $w_{i0} \in O^*$ is the initial multiset of symbols; $w_i \in O^*$ is the current multiset of symbols; R_i is a finite ordered set of multiset rewriting rules, further described below.

The general form of a rule, which transforms state S to state S' is

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

where:

- S, S' are states, $S, S' \in Q_i$, which is a finite set of *states*;
- x, x', y, z, z' are strings, $x, x', y, z, z' \in O^*$, which represent *multisets* of symbols;
- α is a *rewriting* operator, $\alpha \in \{\texttt{min}, \texttt{max}\}$;
- β is a *transfer* operator, $\beta \in \{\uparrow_\gamma, \downarrow_\gamma, \updownarrow_\gamma \mid \gamma \in \{\forall, \exists\} \cup \Lambda\}$, where $\gamma = \forall$ is the default and Λ is the set of (implicit or explicit) arc labels.

All cells evolve *synchronously* (in "lock-step"). A cell evolves by applying one or more rules, which may change its content and state and may send symbols to its neighbours. Given a cell configuration, (Q_i, S_i, w_i, R_i), a rule $S\ x \rightarrow_\alpha S'\ x'\ (u)\beta \mid z \neg z' \in R_i$ is applicable if $S = S_i$, $xz \subseteq w_i$ and $z' \cap w_i = \emptyset$. The application of a rule has two sub-steps: (i) its left hand-side is processed, x is removed from the current content, the next state S' is decided and (ii) its right hand-side is processed, the cell transits to the decided target state S', x' is added to the current content and u is sent as specified by the transfer operator β (as further described below). Multiset z is a *promoter* and multiset z' is an *inhibitor*: z enables the rule, z' disables the rule, but z and z' do not otherwise participate in the rule application and remain unchanged [10]. The rules are applied in *weak priority* order [16]: (a) higher priority applicable rules are applied before the lower priority rules, and (b) a lower priority rule can be applied after other rules only if it reaches the same target state.

The transfer operator β's arrow points in the direction of transfer: $\uparrow$—towards parents; $\downarrow$—towards children; $\updownarrow$—in both directions.

The transfer operator β's qualifier, γ, indicates the distribution form: $\forall$—a broadcast (the default); $\exists$—an anycast (nondeterministic); an arc label—a unicast over a specific arc (i.e. to a specific target).

Operator $\alpha \in \{\texttt{min}, \texttt{max}\}$ 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.

Extensions: We use an extended version of the basic P module framework (described above). Specifically, we use a simple form of complex symbols [13], similar to Prolog terms and Lisp tuples, with a crisp and fast encoding and decoding and a simple unification

semantics, e.g., $f(j,i)$, typically abbreviated as $f_{j,i}$. 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*.

Further, we allow high-level *generic rules*, which are *instantiated* using *free variable* matching [1]. A generic rule is identified by using complex symbols and an extended version of the classical rewriting mode, in fact, a combined *instantiation.rewriting* mode, which is one of {`min.min`, `min.max`, `max.min`, `max.max`}, i.e. we consider all possible combinations of (1) an *instantiation* mode in {`min`, `max`} and (2) a *rewriting* mode in {`min`, `max`}.

As an example, consider all possible instantiations of the generic rule ρ_α (rule 3.6 of P Algorithm 1), in a system where cell σ_7 contains multiset $f^2 s'_2 {s'_3}^2$, where α is one of the four extended rewriting modes:

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

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

$$S_{20}\ f\ s'_2\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_2$$
$$S_{20}\ f\ s'_3\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_3$$

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

$$S_{20}\ f\ s'_2\ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_2$$
$$S_{20}\ f\ s'_3\ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_3$$

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

$$S_{20}\ f\ s'_2\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_2$$
$$S_{20}\ f\ s'_3\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_3$$

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

$$S_{20}\ f\ s'_2\ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_2$$
$$S_{20}\ f\ s'_3\ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_3$$

The interpretation of `min.min`, `min.max` and `max.max` modes is straightforward. While other interpretations could be considered, the mode `max.min` 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) and each one of the instantiated rules is applied *once*, if possible.

Complex symbols and generic rules 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.

3 Disjoint Paths

Consider a *digraph*, $G = (V, E)$, a *source* node, $s \in V$, and a *target* node, $t \in V$. 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, $\overline{\pi}^{-1} = \{(v, u) \mid (u, v) \in \overline{\pi}\} \subseteq E^{-1}$. Given two nodes, x and y, an x-to-y path is a path starting from x and ending in y.

The edge- and node-disjoint paths problems look for a *maximum cardinality set* of edge- and node-disjoint s-to-t paths, respectively. A set of paths are *edge-disjoint* or *node-disjoint* if they have no common arc or no common intermediate node, respectively. Node-disjoint paths are also edge-disjoint paths, but the converse is not true. If the disjoint paths are not *simple*, we can always *simplify* them at the end. The edge-disjoint paths problem can be transformed to a maximum flow problem by assigning unit capacity to each edge [8]. Cormen et al. [3] give a detailed presentation of these topics.

Given a set of edge- or node-disjoint paths P, we define $\overline{P}$, as the set of their arcs, $\overline{P} = \cup_{\pi \in P} \overline{\pi} \subseteq E$, and we recall the definition of 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 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 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, then the arc in $\overline{P}$ and its reversal in $\overline{\alpha}$ "cancel" each other (due to zero total flow) and are discarded. The remaining path fragments are relinked to construct an extended set of 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 established paths, P', until no more augmenting paths are found [8].

Figure 1 illustrates this process, on digraph G, with nodes $\{\sigma_i \mid i \in [0, 7]\}$: (a) shows the 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 disjoint path arcs; (c) shows an augmenting path in G_P, $\alpha = \sigma_0.\sigma_3.\sigma_5.\sigma_2.\sigma_6.\sigma_7$, 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 a larger 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, $G_{P'}$.

Typically, an augmenting paths algorithm uses a DFS algorithm [8] or a BFS algorithm [6], which dynamically build *DFS trees* or *BFS trees*, respectively. Intuitively, a *search tree* is built by dynamically evolving *search paths*, which "start" from the source and "try" to reach the target; at any given time, a search path is a prefix of a branch in the search tree. A *successful search path* in the residual graph becomes a new *augmenting path* and is used to increase the number of disjoint paths. Conceptually, this solves the edge-disjoint paths problem.

However, the *node-disjoint* paths require additional refinements—usually by *node splitting* [11]. Each *intermediate* node, v, is split into an *entry* node, v_1, and an *exit*

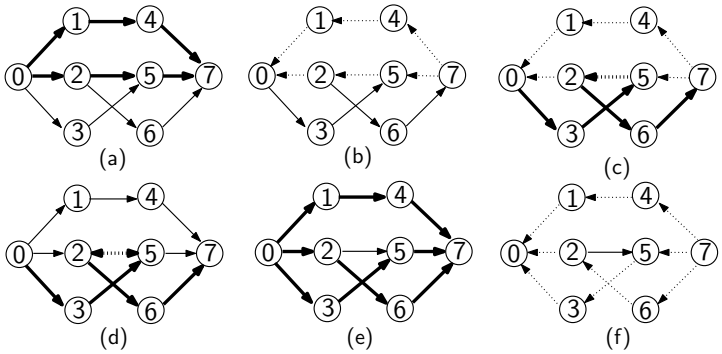


Figure 1: Finding an augmenting path in a residual digraph. Normal arrows: original arcs; thick arrows: disjoint or augmenting path arcs; dotted arrows: reversed path arcs.

node, v_2, linked by an arc (v_1, v_2). Arcs that were directed into v are redirected into v_1 and arcs that were directed out of v are redirected out of v_2. **Figure 2** illustrates this node splitting, where all intermediate nodes are split—this is a bipartite digraph.

4 Disjoint Paths in P Systems

Classical algorithms use the digraph as data and use global information. In contrast, our solutions are *fully distributed*. There is no central cell to convey global information among cells, i.e. cells only communicate with neighbours via local channels (between structural neighbours). Unlike traditional programs, which keep full path information globally, our P systems solutions record paths predecessors and successors locally in each cell, similar to distributed routing tables in networks. Moreover, our cells start without any kind of network knowledge: cells do not know the identities of their children, not even their numbers.

Our DFS-based algorithms start with a preliminary phase, where cells learn the identities of their neighbours. This information is reified as "pointer" symbols: n'_j—indicating that σ_j is a parent; n''_k—indicating that σ_k is a child; d'_j—indicating that σ_j is a disjoint path (dp) predecessor; d''_k—indicating that σ_k is a dp-successor. **Table 1** shows distributed "routing" records for the two disjoint paths of Figure 1 (a). Other digraph paths, e.g. augmented paths and search paths, can be represented in a similar way.

Table 1: Distributed "routing" records for the two disjoint paths of Figure 1 (a).

Cell	σ_0	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7
Parents		n'_0	n'_0	n'_0	n'_1	n'_2 n'_3	n'_2	n'_4 n'_5 n'_6
Children	n''_1 n''_2 n''_3	n''_4	n''_5 n''_6	n''_5	n''_7	n''_7	n''_7	
Dp-predecessors		d'_0	d'_0		d'_1	d'_2		d'_4 d'_5
Dp-successors	d''_1 d''_2	d''_4	d''_5		d''_7	d''_7		

As mentioned, the *node-disjoint* version requires special treatment. Although many P systems accept cell division, we feel that this feature should not be used here and intentionally discard it. Following Dinneen et al. [4], rather than actually splitting P cells,

we simulate this by ad-hoc rules. This approach could be used in other distributed networks, where nodes cannot be split [4]. Essentially, node splitting prevents more than one unit flow through an intermediate node [11].

In our case, node splitting can be simulated by: (i) constraining in and out flow capacities to one and (ii) having two *visited* markers for each intermediate cell, σ_i, one for a *virtual entry* node, σ_i', and another for a *virtual exit* node, σ_i'', extending the visiting idea of classical search algorithms.

As expected, each virtual node can be individually visited at most once in a search round. However, these two virtual nodes are not totally independent, a search path can visit both, but there is one *visiting restriction*, if the search path starts by visiting the exit node. Consider a search path, τ, visiting an intermediate cell, σ_i, which appears in a disjoint path. If τ first visits σ_i'', via a push-back on an outgoing arc, e.g., (σ_i'', σ_j') then it can continue either (a) with another unvisited outgoing arc, e.g., (σ_i'', σ_k'), or (b) with a push-back on the internal virtual arc, (σ_i', σ_i''). In case (b), τ follows by visiting σ_i', while in case (a), τ must be prohibited to visit σ_i', otherwise a loop will occur. There is no restriction, if the search path starts by visiting the entry node.

Figure 2 illustrates a scenario when one cell, σ_2, is visited *twice*, first via its entry and then via its exit node [4]. Assume that we already have a disjoint path, $\pi = \sigma_0.\sigma_1.\sigma_2.\sigma_3.\sigma_4$. Consider a search path, τ, starting from source cell, σ_0, and reaching cell, σ_2, in fact, σ_2's virtual entry node. This is allowed and σ_2's entry node is marked as visited. However, to constrain its in-flow to one, σ_2 can only push-back τ on its in-flow arc, (σ_1, σ_2). Cell σ_1's exit node becomes visited, σ_1's out-flow becomes zero and τ continues on σ_1's outgoing arc, (σ_1, σ_3). When τ reaches cell σ_3, σ_3's entry node becomes visited and σ_3 pushes τ back on its in-flow arc, (σ_2, σ_3). Cell σ_2's virtual exit node becomes visited, σ_2's out-flow becomes zero and τ continues on σ_2's outgoing arc, (σ_2, σ_4). Finally, the search path, τ, reaches the target, σ_4, and becomes $\tau = \sigma_0.\sigma_2.\sigma_1.\sigma_3.\sigma_2.\sigma_4$. After removing cancelling arcs and relinking the remaining arcs, we obtain two node-disjoint paths, $\sigma_0.\sigma_1.\sigma_3.\sigma_4$ and $\sigma_0.\sigma_2.\sigma_4$. Note that, as required by the above visiting restriction, after taking an outgoing arc from the exit node, σ_1'', path τ is prohibited to ever visit the corresponding entry node, σ_1', e.g., via a hypothetical arc (σ_3, σ_1).

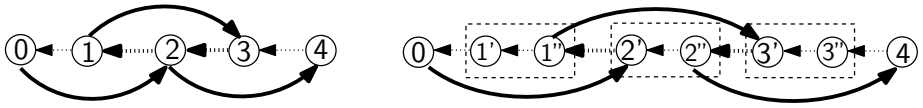


Figure 2: Simulating node splitting for determining node-disjoint paths [4]. Virtual entry nodes are indicated by single quotes and virtual exit nodes by double quotes.

5 Our DFS-based Algorithms

As earlier mentioned, augmenting paths can be searched using depth-first search (DFS) or breadth-first search (BFS). Conceptually, DFS explores as far as possible along a

single branch, before backtracking, while BFS explores as many branches as possible concurrently. P systems can exploit BFS's massive parallelism and also the limited, but not insignificant, parallelism possible in DFS.

Dinneen et al. proposed two classical DFS-based algorithms [4], here called DKN-DFS-Edge and DKN-DFS-Node, which find edge- and node-disjoint paths. These algorithms were originally designed for (undirected) graphs; however, in this paper we consider slightly modified versions that work on digraphs. We focus here on DKN-DFS-Edge; but most of the following discussion also applies to DKN-DFS-Node, which is based on DKN-DFS-Edge.

The appendix presents the main algorithms discussed in this paper, as high-level **pseudocode**, stated in terms of *cells* (instead of *nodes*). The distributed algorithms have been *sequentialized*: this improves their readability, but also sequentializes actions which actually run in parallel on different cells.

1. Classical edge-disjoint paths algorithm, based on augmenting paths
2. Classical DFS algorithm (sequential version)
3. Classical edge-disjoint path algorithm, after unrolling its first DFS call
4. Sequentialized version of Dinneen et al.'s DKN-DFS-Edge algorithm
5. Sequentialized version of our NW-DFS-Edge algorithm
6. Sequentialized Cidon's DFS algorithm

The first two algorithms are well-known and the third is a straightforward transformation, so we do not insist on these.

Algorithm 1: Classical edge-disjoint paths
As shown by Pseudocode 1, this is the classical edge-disjoint paths algorithm, based on the well-known Ford-Fulkerson's max-flow algorithm. To find augmenting paths, it uses the classical DFS algorithm, Algorithm 2. 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 cells and arcs as unvisited after each new augmenting path, until no more augmenting paths are found.

Algorithm 2: Classical DFS
Pseudocode 2 presents the sequential version of the classical DFS algorithm. This algorithm has also a distributed version [18], which is actually used by Algorithm 4. The distributed version records "bread-crumbing" visited information by marking both cells and arcs.

Algorithm 3: Classical edge-disjoint paths–1st DFS call unrolled
As shown by Pseudocode 3, this algorithm is a straightforward transformation of Algorithm 1, by unrolling its first DFS call.

Algorithm 4: DKN-DFS-Edge
Essentially, algorithm DKN-DFS-Edge is a faster version of Algorithm 3, which limits

the probing of source cell's children. Its sequentialized version is shown by Pseudocodes 4 and 2. DKN-DFS-Edge uses a bounded **for** loop over the source cell's children, never probing previously probed children, although these are reset as unvisited after a successful round [4] (this idea seems part of algorithm folklore, but the authors have not been able to track it to a formal source).

Algorithm DKN-DFS-Edge starts with an empty set of disjoint paths, $P_0 = \emptyset$, a trivial residual graph, $G_0 = (V, E_0)$, where $E_0 = E$ (i.e. $G_0 = G$),

The algorithm works in d successive *search rounds*, defined by successive *iterations* of the **for** loop, where: r is the round number, $r \in [1, d]$; d is the outdegree of the starting cell σ_s; P_{r-1} is the set of disjoint paths available at the start of round $\#r$; and $G_{r-1} = (V, E_{r-1})$ is the residual digraph available at the start of round $\#r$ Without loss of generality, we assume that σ_s's children are represented by the set $\{\sigma_{s1}, \sigma_{s2}, \ldots, \sigma_{sd}\}$.

Search round $\#r$ attempts to find a new augmenting path, starting with $\sigma_s.\sigma_{sr}$, using the classical DFS Algorithm 2, on the current residual digraph, $G_{r-1} = (V, E_{r-1})$.

Search round $\#r$ starts when σ_s sends a *visit token* to σ_{sr} (lines 4.4–5): the receiving child, σ_{sr}, becomes the search "frontier". A current frontier cell *forwards* the visit token over an arbitrarily selected unvisited arc (lines 2.8–9). An unvisited cell accepts this token and becomes the new frontier, by inserting itself at the front of the current search path (line 2.6). A previously visited cell (line 2.5) or a cell which does not have any (more) unvisited arcs (line 2.11) sends back a *backtrack token* (line 2.12), to return the frontier to its search path predecessor.

If the search path reaches the *target* cell, σ_t, then round $\#r$ is *successful*: an augmenting path, α_r, is found and σ_t sends a *path confirmation* back to σ_s. A successful round $\#r$ increments the set of disjoint paths: while moving towards σ_s, the confirmation reshapes the existing disjoint paths and the newly found augmenting path, α_r, by discarding cancelling arcs and relinking the rest, building a larger set of disjoint paths, P_r, $|P_r| = |P_{r-1}| + 1$ and a new residual digraph, $G_r = (V, E_r)$. Thus, lines 4.10–12 of Pseudocode 4 are actually done within Pseudocode 2, during the return from a successful search.

After receiving the path confirmation, σ_s initiates a global *reset* broadcast, carried by a *reset token*, which resets all visited cells and arcs to unvisited, reinitialising them for the next search round. In our distributed implementation, the reset process, line 4.13 of Pseudocode 4, runs in parallel with the next round, without affecting it: the reset starts two steps ahead of the next round and progresses with the same speed, keeping its distance.

If the search path cannot reach σ_t, then eventually the backtrack token arrives at the source, σ_s, and the round *fails* (line 4.7). A failed round does *not* change the current set of disjoint path or the current residual digraph, i.e. $P_r = P_{r-1}$, $G_r = G_{r-1}$ (i.e. $E_r = E_{r-1}$), and does *not* reset the visited cells and arcs, offering some speed improvements (line 4.8).

Although not explicit in Pseudocode 4, after probing all its children, the source cell, σ_s, initiates a global broadcast, carried by a *finalise token*. This finalisation is not strictly necessary, but informs all cells that the algorithm has terminated, clears their contents and brings them to their final states.

The correctness of DKN-DFS-Edge is ensured by Propositions 2, 3, 4 and 5 (see [4]

for proofs).

Proposition 2. Round $\#r$ of DKN-DFS-Edge succeeds iff the residual digraph G_r has at least one augmenting path starting with $\sigma_s.\sigma_{sr}$.

Proposition 3. A successful DKN-DFS-Edge round $\#r$ returns one augmenting path in the residual digraph G_r and a new disjoint path in the original digraph G, both starting with with $\sigma_s.\sigma_{sr}$.

Proposition 4. If round $\#r$ of DKN-DFS-Edge fails, then none of the residual digraphs $G_{r'}$, $r' > r$, has any augmenting path starting with $\sigma_s.\sigma_{sr}$.

Proposition 5. The DKN-DFS-Edge algorithm finds a maximum cardinality set of edge disjoint paths.

We propose two new algorithms, NW-DFS-Edge and NW-DFS-Node, which further improve DKN-DFS-Edge and DKN-DFS-Node, respectively, using:

1. ideas from Cidon's distributed DFS [2], adapted for the synchronous case. When a cell is visited for the first time, in a round, it sends *visited notifications* to all its neighbours, concurrently with the visit token. Thus, cells can avoid forwarding the visit token to cells that were previously visited, in the same round. Essentially, this improvement speeds up DFS from $O(m)$ to $O(n)$, if a limited form of parallelism is available.

2. Theorem 6, a *generalisation* of Proposition 4, which shows that cells visited during a failed round can be ignored in any following round. Conceptually, these cells visited during a failed round remain forever in a *permanently visited* state.

Leveraging the parallel and distributed features of P systems, the housekeeping operations required by these two optimisations can be performed concurrently with the main search.

Theorem 6. If round $\#r$ of DKN-DFS-Edge fails, after visiting σ_f, then none of the residual digraphs $G_{r'}$, $r' > r$, has any augmenting path containing σ_f.

Proof. The proof is a thought experiment which leads to a contradiction.

Consider a complete run of DKN-DFS-Edge, which successively probes each of σ_s's children, $\sigma_{s1}, \sigma_{s2}, \ldots \sigma_{sd}$, and determines a set of edge-disjoint paths: $\pi_j = \sigma_s.\sigma_{sj}.\pi'_j.\sigma_t$, $j \in Q$, $Q \subseteq [1, d]$.

Consider two rounds, $\#r$ and $\#r'$, $r < r'$, which visit cell σ_f. Assume, by contradiction, that (a) round $\#r$ fails, but (b) round $\#r'$ succeeds in finding an augmenting path containing σ_f. Thus, (a) digraph G_r has a path $\tau = \sigma_s.\sigma_{sr}.\delta.\sigma_f$ and (b) digraph $G_{r'}$ has an augmenting path $\alpha = \sigma_s.\sigma_{sr'}.\alpha'.\sigma_{f'}.\sigma_f.\sigma_{f''}.\alpha''.\sigma_t$. Moreover, assume that round $\#r'$ is the *first* successful round revisiting σ_f, after round $\#r$.

As a thought experiment, consider starting the same algorithm on digraph $G' = (V, E')$, where $E' = E_{r-1} \setminus \{(\sigma_s, \sigma_{si}), (\sigma_{si}, \sigma_s) \mid i \in [1, r-1]\}$, thus G' is the residual digraph G_{r-1}, after removing all arcs between σ_s and its first $r-1$ original children.

Clearly, the absence of these arcs will not affect any of following rounds: round $r-1+p$ on digraph G and round p on digraph G', for $p \in [1, d-r+1]$, follow exactly the same paths and return exactly the same results.

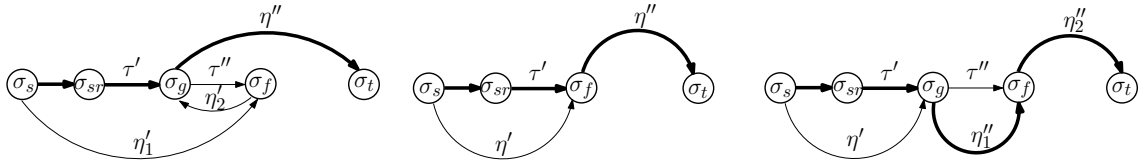
Thus, on G', (a) round #1 visits cell σ_f, by search path τ, and then fails and (b) round $\#r''$, $r'' = r' - r + 1$, is the first successful round, after round #1, revisiting σ_f on an augmenting path, α.

Because τ appears in the *first* round, all its arcs belong to G', $\overline{\tau} \subseteq E'$. Because α is the *first* augmenting path passing through σ_f, $\{(\sigma_{f'}, \sigma_f), (\sigma_f, \sigma_{f''})\} \subseteq E'$. Moreover, one of the new edge disjoint paths, determined at round $\#r''$, will also pass through σ_f: $\theta = \sigma_s.\theta'.\sigma_{f'}.\sigma_f.\sigma_{f''}.\theta''.\sigma_t$; and, of course, all θ's arcs (like all disjoint paths arcs) belong to G', $\overline{\theta} \subseteq E'$.

Paths τ and θ have at least one common cell, σ_f. Let σ_g be the first cell on τ that appears on θ, then $\tau = \sigma_s.\sigma_{sr}.\tau'.\sigma_g.\tau''.\sigma_f$ (or $\tau = \sigma_s.\sigma_{sr}.\tau'.\sigma_f$, if $\sigma_g = \sigma_f$) and $\theta = \sigma_s.\eta'.\sigma_g.\eta''.\sigma_t$. Then, combining the first part of τ (up to σ_g) with the second part of θ (after σ_g), we obtain a σ_s-to-σ_t path in G': $\mu = \sigma_s.\sigma_{sr}.\tau'.\sigma_g.\eta''.\sigma_t$. Figure 3 illustrates this construction, in three different cases, according to the relative position of σ_f on θ (σ_f before σ_g, $\sigma_f = \sigma_g$, σ_f after σ_g).

To summarize, σ_t is reachable by a path in G', which starts with $\sigma_s.\sigma_{sr}$. Therefore, any DFS tree on G', rooted at σ_{sr}, has a branch reaching σ_t. Thus, round #1 on G', should have found an augmenting path and a disjoint path, starting with $\sigma_s.\sigma_{sr}$; briefly, it should have succeeded (see also Propositions 2 and 3).

This contradicts the assumptions that round #1 on G' and round $\#r$ on G fail. □



(a) σ_f before σ_g
$\eta' = \eta'_1.\sigma_f.\eta'_2$
$\tau = \sigma_s.\sigma_{sr}.\tau'.\sigma_g.\tau''.\sigma_f$
$\theta = \sigma_s.\eta'_1.\sigma_f.\eta'_2.\sigma_g.\eta''.\sigma_t$
$\mu = \sigma_s.\sigma_{sr}.\tau'.\sigma_g.\eta''.\sigma_t$

(b) $\sigma_f = \sigma_g$
$\tau = \sigma_s.\sigma_{sr}.\tau'.\sigma_f$
$\theta = \sigma_s.\eta'.\sigma_f.\eta''.\sigma_t$
$\mu = \sigma_s.\sigma_{sr}.\tau'.\sigma_f.\eta''.\sigma_t$

(c) σ_f after σ_g
$\eta'' = \eta''_1.\sigma_f.\eta''_2$
$\tau = \sigma_s.\sigma_{sr}.\tau'.\sigma_g.\tau''.\sigma_f$
$\theta = \sigma_s.\eta'.\sigma_g.\eta''_1.\sigma_f.\eta''_2.\sigma_t$
$\mu = \sigma_s.\sigma_{sr}.\tau'.\sigma_g.\eta''_1.\sigma_f.\eta''_2.\sigma_t$

Figure 3: The augmenting path, μ, constructed by combining the first part of search path τ and the second part of disjoint path θ.

Proposition 4, which was not used in this proof, appears now as a corollary of Theorem 6, for the particular case when σ_f is a child of σ_s.

Pseudocode 5 shows the sequential version of pseudocode for NW-DFS-Edge, which is the same as Pseudocode 4 except lines 4–7, 9–10, 13, 18. It uses the modified Cidon's DFS (Pseudocode 6).

Based on Theorem 6, visited cells in failed rounds can be ignored in all following rounds. In our NW-DFS-Edge algorithm, the source cell, σ_s broadcasts two kinds of end-of-round resets: (1) *after-success* resets, which change all *temporarily* visited marks to

unvisited (line 5.18) and (2) *after-failure* resets, which logically remove cells *temporarily* visited in a failed round from further operations (line 5.13).

5.1 Rules for DFS-based Edge-disjoint Paths

The P system rules for edge- and node-disjoint paths are slightly different, due to the simulated node-splitting approach, but the basic principle is the same. Figure 4 shows how a cell becomes a frontier cell and sends the visit or backtrack token in our DFS-based edge- and node-disjoint algorithms.

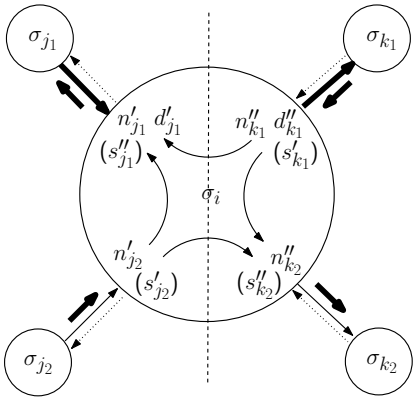


Figure 4: How a cell, σ_i, becomes a frontier cell and sends the visit or backtrack token. Dotted arcs indicate optional symmetric arcs. Thick arcs indicate disjoint path arcs. Thick arrows near arcs indicate visit tokens. Symbols in parentheses are generated after σ_i becomes a frontier cell and sends the visit or backtrack token.

We use the following abbreviations: dp-predecessor—disjoint path predecessor; dp-successor—disjoint path successor; sp-predecessor—search path predecessor; sp-successor—search path successor.

Cell σ_i uses the following symbols to record its relationships with a neighbouring cell σ_j: n'_j indicates a parent; n''_k indicates a child; d'_j indicates a dp-predecessor; d''_k indicates a dp-successor; s'_j indicates a sp-predecessor; s''_k indicates a sp-successor. Cell σ_i uses a single symbol, f_i, for both its forward and push-back visit tokens.

There are three cases how a cell, σ_i, becomes a frontier cell:

(Ia) unvisited σ_i receives a forward visit token, f_{j_2}, from its parent, σ_{j_2}, indicated by n'_{j_2}, and records its sp-predecessor as s'_{j_2};

(Ib) unvisited σ_i receives a push-back visit token, f_{k_1}, from its dp-successor, σ_{k_1}, indicated by d''_{k_1}, and records its sp-predecessor as s'_{k_1};

(Ic) visited σ_i receives a backtrack token, b_l, from its sp-successor, σ_l, indicated by s''_l, and erases s''_l (σ_l and s''_l are not shown in Figure 4).

There are four cases how a frontier cell, σ_i, sends the forward or push-back visit token, or backtrack token:

(IIa) if σ_i has any unvisited child, σ_{k_2}, indicated by n''_{k_2} and the visiting notification from σ_{k_2}, σ_i sends a forward visit token, f_i, to σ_{k_2}, and records its sp-successor as s''_{k_2};

(IIb) otherwise, if σ_i has an unvisited dp-predecessor, σ_{j_1}, indicated by d'_{j_1} and the visiting notification from σ_{j_1}, σ_i sends a push-back visit token, f_i, to σ_{j_1}, and records its sp-successor as s''_{j_1};

(IIc) otherwise, if σ_i has no unvisited child or unvisited dp-predecessor, σ_i sends a backtrack token, b_i, to its sp-predecessor, indicated by s'_{j_2} or s'_{k_1}, and erases s'_{j_2} or s'_{k_1}.

We merge case (Ia) with (Ib), to refactor the rules, using a single symbol for both the forward and push-back tokens. As shown in Figure 4, for case (IIa), cell σ_i can not send the forward visit token to its dp-successor, σ_{k_1}. We also handle the symmetric arc case: cell σ_i does not send the forward visit token to its child, indicated by n''_{j_1}, if σ_{j_1} is its dp-predecessor, indicated by d'_{j_1}. Note that, for the node-disjoint path algorithm, case (IIa) is only allowed when σ_i does not appear in a disjoint path, to constrain the in-flow to one.

P Algorithm 1: NW-DFS-Edge

Input: All cells start with the same set of rules and without any topological awareness (they do not even know their local neighbours). 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_{50}. On completion, all cells are *empty*, with the following 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 *dp-predecessors*, d'_j, and *dp-successors*, d''_k.

Table 2 shows the initial and final configurations of the P system defined by P Algorithm 1, for Figure 1.

The right column of the following P rules shows the lines of Pseudocodes 5 and 6 in the appendix section, in the form of #pseudocode.#line.

Table 2: Initial and final configurations of P Algorithm 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	$S_{50}\ \iota_0\ s\ d''_1\ d''_2\ d''_3$	$S_{50}\ \iota_1\ d'_0\ d''_4$	$S_{50}\ \iota_2\ d'_0\ d''_6$	$S_{50}\ \iota_3\ d'_0\ d''_5$
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	$S_{50}\ \iota_4\ d'_1\ d''_7$	$S_{50}\ \iota_5\ d'_3\ d''_7$	$S_{50}\ \iota_6\ d'_2\ d''_7$	$S_{50}\ \iota_7\ t\ d'_4\ d'_5\ d'_6$

1. Initial differentiation (S_1–S_3)

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. $S_0\ \rightarrow_{\texttt{min}}\ S_0\ n \mid s$
3. $S_1\ \rightarrow_{\texttt{min}}\ S_2$
4. $S_2\ n\ \rightarrow_{\texttt{max}}\ S_3$
5. $S_2\ \rightarrow_{\texttt{min}}\ S_3$
6. $S_3\ \rightarrow_{\texttt{min}}\ S_{10}\ f \mid s$
7. $S_3\ \rightarrow_{\texttt{min}}\ S_{30} \mid t$
8. $S_3\ \rightarrow_{\texttt{min}}\ S_{20}$

2. Source cell (S_{10})

1. $S_{10}\ f\ \rightarrow_{\texttt{min.min}}\ S_{10}\ s''_k\ (v_i)\updownarrow_\forall\ (f_i)\downarrow_k\ \mid n''_k \neg v_k$ — line 5.4–7, 5.9
2. $S_{10}\ a\ s''_k\ n''_k\ \rightarrow_{\texttt{min.min}}\ S_{40}\ r\ d''_k\ (r)\updownarrow$ — line 5.18
3. $S_{10}\ b_k\ s''_k\ n''_k\ \rightarrow_{\texttt{min.min}}\ S_{40}\ q\ (q)\updownarrow$ — line 5.13
4. $S_{10}\ f\ \rightarrow_{\texttt{min.min}}\ S_{50}\ (g)\updownarrow$ — line 5.20

3. Frontier cell (S_{20})

1. $S_{20}\ f_j\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v\ s'_j\ (v_i)\updownarrow_\forall\ f \mid \iota_i \neg v$ — lines 6.5–8
2. $S_{20}\ f_j\ \rightarrow_{\texttt{min.min}}\ S_{20}\ (b_i)\updownarrow_j\ \mid v\ \iota_i$ — lines 6.9–13
3. $S_{20}\ b_k\ s''_k\ \rightarrow_{\texttt{min.min}}\ S_{20}\ f$ — lines 6.9–13
4. $S_{20}\ f\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v_k\ s''_k\ (f_i)\downarrow_k\ \mid \iota_i\ n''_k \neg v_k\ d'_k\ d''_k$ — lines 6.9–13
5. $S_{20}\ f\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v_k\ s''_k\ (f_i)\uparrow_k\ \mid \iota_i\ d'_k \neg v_k$ — lines 6.9–13
6. $S_{20}\ f\ s'_j\ \rightarrow_{\texttt{min.min}}\ S_{20}\ (b_i)\updownarrow_j\ \mid \iota_i$ — line 6.14

4. Intermediate cell (S_{20}–S_{21})

1. $S_{20}\ g\ \rightarrow_{\texttt{min.min}}\ S_{50}\ (g)\updownarrow$ line 5.20
2. $S_{20}\ v_k\ \rightarrow_{\texttt{max}}\ S_{20} \mid v_k$
3. $S_{20}\ a\ s'_j\ s''_k\ \rightarrow_{\texttt{min.min}}\ S_{20}\ d\ d'_j\ d''_k\ (a)\updownarrow_j$ lines 5.15–17
4. $S_{20}\ d\ d''_k\ d'_k\ \rightarrow_{\texttt{min.min}}\ S_{20}$ lines 5.15–17
5. $S_{20}\ \rightarrow_{\texttt{min}}\ S_{21}\ (q)\updownarrow_\forall\ \mid q$ line 5.13
6. $S_{20}\ \rightarrow_{\texttt{min}}\ S_{21}\ w \mid q\ v \neg w$ line 5.13
7. $S_{20}\ \rightarrow_{\texttt{max.min}}\ S_{21}\ w_k \mid q\ v_k \neg w_k$ line 5.13
8. $S_{20}\ \rightarrow_{\texttt{min}}\ S_{21}\ (r)\updownarrow_\forall\ \mid r$ line 5.18
9. $S_{20}\ v\ \rightarrow_{\texttt{min}}\ S_{21} \mid r \neg w$ line 5.18
10. $S_{20}\ v_k\ \rightarrow_{\texttt{max.min}}\ S_{21} \mid r \neg w_k$ line 5.18
11. $S_{21}\ \rightarrow_{\texttt{min}}\ S_{40}$

5. Target cell (S_{30})

1. $S_{30}\ g\ \rightarrow_{\texttt{min.min}}\ S_{40}\ (g)\updownarrow$ line 5.20
2. $S_{30}\ f_j\ \rightarrow_{\texttt{min.min}}\ S_{30}\ d'_j\ (a)\downarrow_j$ line 6.4
3. $S_{30}\ \rightarrow_{\texttt{min}}\ S_{40} \mid q$ line 5.13
4. $S_{30}\ \rightarrow_{\texttt{min}}\ S_{40} \mid r$ line 5.18

6. All cells (S_{40}: at the end of each round; S_{50}: at the end of the algorithm.)

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. $S_{40}\ a\ \rightarrow_{\texttt{max}}\ S_{40}$
4. $S_{40}\ q\ \rightarrow_{\texttt{max}}\ S_{40}$
5. $S_{40}\ r\ \rightarrow_{\texttt{max}}\ S_{40}$
6. $S_{40}\ \rightarrow_{\texttt{min}}\ S_3$
7. $S_{50}\ g\ \rightarrow_{\texttt{max}}\ S_{50}$
8. $S_{50}\ n'_j\ \rightarrow_{\texttt{max.min}}\ S_{50}$

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

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

11. $S_{50}\ w\ v\ \rightarrow_{\texttt{max}}\ S_{50}$

Figure 5 shows the state transitions of P Algorithm 1.

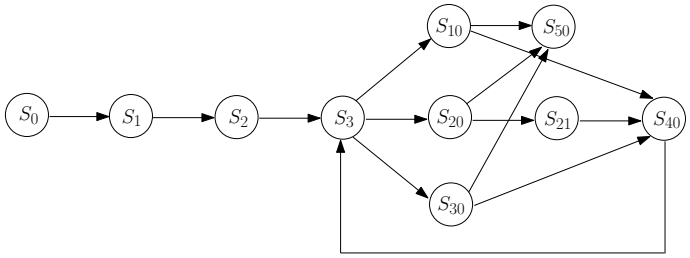


Figure 5: State-chart of our DFS-based algorithms.

This algorithm has two phases:

1. Phase I (rules for initial differentiation), which builds neighbourhood knowledge and prompts each cell to enter its corresponding state:
   - Source cell, σ_s: S_{10}
   - Target cell, σ_t: S_{30}
   - Intermediate cells, σ_i: S_{20}
2. Phase II (rules specifically for the source cell, frontier cells, intermediate cells and the target cell, and common rules for all cells), which searches disjoint paths.

In Phase I, the source cell, σ_s generates one symbol, n. This simulates that σ_s receives a symbol n from a non-existing cell (rule 1.2). On receiving one or more n's, each cell, σ_i, discards one n, sends its ID to (a) its parents, encoded as n''_i, and (b) to its children, encoded as n'_i, and further broadcasts n, to all its neighbours (rule 1.1), Any superfluous n's are discarded after one step (rule 1.4). Next, each cell enters its corresponding state: (1) the source cell, σ_s, indicated by s, enters S_{10} (rule 1.6); (2) the target cell, σ_t, indicated by t, enters S_{30} (rule 1.7); (3) intermediate cells, σ_i, enter S_{20} (rule 1.8).

In Phase II, the following symbols are sent by cell σ_i: f_i is the visit token; b_i is the backtrack token; v_i is the visiting notification; a is the path confirmation; q is the after-success reset token; r is the after-failure reset token; g is the finalise token.

Also, cell σ_i uses the following symbols to record its state: v indicates that it is visited; f indicates that it is the frontier cell; d indicates that it appears on a disjoint path.

For example, **Figure 1 (c)** shows a disjoint path, $\sigma_0.\sigma_2.\sigma_5.\sigma_7$, and an augmenting path, $\sigma_0.\sigma_3.\sigma_5.\sigma_2.\sigma_6.\sigma_7$. **Table 3** shows cells' contents at stage (c). Cell σ_5 contains: a dp-predecessor, d'_2, a dp-successor, d''_7, a sp-predecessor, s'_3 and a sp-successor, s''_2. Cell σ_2 contains: a dp-predecessor, d'_0, a dp-successor, d''_5, a sp-predecessor, s'_5 and a sp-successor, s''_6.

The evolution of the source cell: In each round, the source cell, σ_s, starts the search by sending its visit token, f_s, to one of its unvisited children, σ_k, indicated by n''_k and inhibitor v_k, and broadcasts its visiting notification, v_s, to all its neighbours (rule 2.1). If σ_s receives a path confirmation, a, it transforms its sp-successor, s''_k, into a dp-successor, d''_k, deletes the child symbol, n''_k, and initiates a global broadcast, carried by an after-success reset token, r (rule 2.2). Otherwise, if σ_s receives a backtrack token, b_k, it deletes s''_k and n''_k, applying Proposition 4, and initiates a global broadcast, carried by an after-failure reset token, q (rule 2.3). After probing all its children, σ_s initiates a global broadcast, carried by a finalise token, g (rule 2.4).

The evolution of frontier cells: As mentioned in the explanations of Figure 4, an unvisited intermediate cell, σ_i, becomes a frontier cell, by receiving a (forward or push-back) visit token, f_j, refactoring rules for cases (Ia) and (Ib) (rule 3.1). Cell σ_i records its sp-predecessor as s'_j, marks itself as visited by v, and generates one f, which indicates that it is the frontier cell.

As in case (Ic), a visited intermediate cell, σ_i, becomes a frontier cell, by receiving a backtrack token, b_k, from its sp-successor, s''_k (rule 3.3).

Using an idea from **Cidon's distributed DFS**, σ_i does not attempt to visit any visited σ_k, indicated by an inhibitor, v_k. As in case (IIa), if σ_i has any unvisited child, indicated by n''_k and inhibitor v_k, and σ_k is neither a dp-predecessor, d'_k, nor a dp-successor, d''_k, it sends the visit token, f_i, to σ_k (rule 3.4). Otherwise, as in case (IIb), if σ_i has an unvisited dp-predecessor, indicated by d'_k and inhibitor v_k, it sends the visit token, f_i, to σ_k (rule 3.5). Otherwise, as in case (IIc), σ_i sends its backtrack token, b_i, to its sp-predecessor, indicated by s'_j, and erases s'_j (rule 3.6).

The evolution of other intermediate cells: Consider the case of a former frontier cell, σ_i, receiving a path confirmation, a. Cell σ_i generates one d, and transforms its sp-predecessor, s'_j, and sp-successor, s''_j, into a dp-predecessor, d'_j, and a dp-successor, d''_k, respectively (rule 4.3). Cell σ_i may already contain (from a previous round) a dp-predecessor and a sp-successor, $d'_{j'}$, $d''_{k'}$. If $j = k'$, then d'_j and $d''_{k'}$ are deleted, as one end of the cancelling arc pair, (j, i) and (i, j); similarly, if $k = j'$, then d''_k and $d'_{j'}$ are deleted (rule 4.4).

To apply the optimisation based on **Theorem 6**, at the end of a round, on receiving the after-failure reset token, q, σ_i generates w for v, and w_k's for v_k's (rules 4.6–7), to make them permanently visited. On receiving the after-success reset token, r, σ_i reset visited neighbours to unvisited, by deleting the v and v_k's, which have no corresponding w and w_k's (rules 4.9–10).

The evolution of the target cell; On receiving a visit token, f_j, the target cell, σ_t, records its dp-predecessor, d'_j, and sends back a path confirmation, a (rule 5.2).

Table 3 shows P Algorithm 1 trace fragments for stages (a), (c) and (e) of Figure 1. Symbols, d'_j and d''_k, indicate dp-predecessors and dp-successors, respectively; symbols, s'_j and s''_k, indicate sp-predecessors and sp-successors, respectively. In stage 1(a), the two disjoint paths, $\sigma_0.\sigma_1.\sigma_4.\sigma_7$ and $\sigma_0.\sigma_2.\sigma_5.\sigma_7$, are recorded by the following cell contents: $\sigma_0 : \{d''_1, d''_2\}$, $\sigma_1 : \{d'_0, d''_4\}$, $\sigma_2 : \{d'_0, d''_5\}$, $\sigma_4 : \{d'_1, d''_7\}$, $\sigma_5 : \{d'_2, d''_7\}$, $\sigma_7 : \{d'_4, d'_5\}$. In stage 1(c), the successful search path $\sigma_0.\sigma_3.\sigma_5.\sigma_2.\sigma_6.\sigma_7$ is recorded as: $\sigma_0 : \{s''_3\}$, $\sigma_3 : \{s'_0, s''_5\}$, $\sigma_5 : \{s'_3, s''_2\}$, $\sigma_2 : \{s'_5, s''_6\}$, $\sigma_6 : \{s'_2, s''_7\}$, $\sigma_7 : \{d'_6\}$ (the target records d'_6 directly). In stage 1(e), there are three disjoint paths, $\sigma_0.\sigma_1.\sigma_4.\sigma_7$, $\sigma_0.\sigma_2.\sigma_6.\sigma_7$ and $\sigma_0.\sigma_3.\sigma_5.\sigma_7$, which

are recorded as: $\sigma_0 : \{d''_1, d''_2, d''_3\}$, $\sigma_1 : \{d'_0, d''_4\}$, $\sigma_2 : \{d'_0, d''_6\}$, $\sigma_3 : \{d'_0, d''_5\}$, $\sigma_4 : \{d'_1, d''_7\}$, $\sigma_5 : \{d'_3, d''_7\}$, $\sigma_6 : \{d'_2, d''_7\}$, $\sigma_7 : \{d'_4, d'_5, d'_6\}$.

Table 3: P Algorithm 1 trace fragments for stages (a), (c) and (e) of Figure 1.

Stage	σ_0	σ_1	σ_2	σ_3
1(a)	$S_{10}\ \iota_0\ s\ d''_1\ d''_2\ \ldots$	$S_{20}\ \iota_1\ d'_0\ d''_4\ \ldots$	$S_{20}\ \iota_2\ d'_0\ d''_5\ \ldots$	$S_{10}\ \iota_3\ \ldots$
1(c)	$S_{10}\ \iota_0\ s\ d''_1\ d''_2\ s''_3\ v_2\ v_3\ \ldots$	$S_{20}\ \iota_1\ d'_0\ d''_4\ \ldots$	$S_{20}\ \iota_2\ d'_0\ d''_5\ s'_5\ s''_6\ v\ v_0\ v_5\ \ldots$	$S_{20}\ \iota_3\ s'_0\ s''_5\ v\ v_0\ v_5\ \ldots$
1(e)	$S_{10}\ \iota_0\ s\ d''_1\ d''_2\ d''_3$	$S_{20}\ \iota_1\ d'_0\ d''_4$	$S_{20}\ \iota_2\ d'_0\ d''_6$	$S_{20}\ \iota_3\ d'_0\ d''_5$
Stage	σ_4	σ_5	σ_6	σ_7
1(a)	$S_{20}\ \iota_4\ d'_1\ d''_7\ \ldots$	$S_{20}\ \iota_5\ d'_2\ d''_7\ \ldots$	$S_{20}\ \iota_6\ \ldots$	$S_{30}\ \iota_7\ t\ d'_4\ d'_5\ \ldots$
1(c)	$S_{20}\ \iota_4\ d'_1\ d''_7\ \ldots$	$S_{20}\ \iota_5\ d'_2\ d''_7\ s'_3\ s''_2\ v\ v_2\ v_3\ldots$	$S_{20}\ \iota_6\ s'_2\ s''_7\ v\ v_2\ v_7\ \ldots$	$S_{30}\ \iota_7\ t\ d'_4\ d'_5\ d'_6\ v_5\ v_6\ \ldots$
1(e)	$S_{20}\ \iota_4\ d'_1\ d''_7$	$S_{20}\ \iota_5\ d'_3\ d''_7$	$S_{20}\ \iota_6\ d'_2\ d''_7$	$S_{30}\ \iota_7\ t\ d'_4\ d'_5\ d'_6$

The following proposition encapsulates the above arguments:

Proposition 7. When P Algorithm 1 ends, disjoint path predecessors and successors symbols in its output indicate a maximal cardinality set of edge-disjoint paths.

5.2 Rules for DFS-based Node-disjoint Paths

P Algorithm 2: NW-DFS-Node

Input: As in the edge-disjoint paths algorithm of P Algorithm 1.

Output: Similar to the edge-disjoint paths algorithm. However, the predecessor and successor symbols indicate *node-disjoint paths*, instead of edge-disjoint paths.

Table 4 shows the initial and final configurations of the P system defined by P Algorithm 2, for Figure 6.

Table 4: Initial and final configurations of Algorithms 2, for Figure 6.

Cell	σ_0	σ_1	σ_2	σ_3	σ_4	σ_5
Initial	$S_0\ \iota_0\ s$	$S_0\ \iota_1$	$S_0\ \iota_2$	$S_0\ \iota_3$	$S_0\ \iota_4$	$S_0\ \iota_5$
Final	$S_{50}\ \iota_0\ s\ d''_1\ d''_4$	$S_{50}\ \iota_1\ d'_0\ d''_8$	$S_{50}\ \iota_2\ d'_5\ d''_6$	$S_{50}\ \iota_3\ d'_9\ d''_{10}$	$S_{50}\ \iota_4\ d'_0\ d''_5$	$S_{50}\ \iota_5\ d'_4\ d''_2$
Cell	σ_6	σ_7	σ_8	σ_9	σ_{10}	
Initial	$S_0\ \iota_6$	$S_0\ \iota_7$	$S_0\ \iota_8$	$S_0\ \iota_9$	$S_0\ \iota_{10}\ t$	
Final	$S_{50}\ \iota_6\ d'_2\ d''_7$	$S_{50}\ \iota_7\ d'_6\ d''_{10}$	$S_{50}\ \iota_8\ d'_1\ d''_9$	$S_{50}\ \iota_9\ d'_8\ d''_3$	$S_{50}\ \iota_{10}\ t\ d'_3\ d'_7$	

1. Initial differentiation (S_1–S_3)

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. $S_0\ \rightarrow_{\texttt{min}}\ S_0\ n \mid s$
3. $S_1\ \rightarrow_{\texttt{min}}\ S_2$
4. $S_2\ n\ \rightarrow_{\texttt{max}}\ S_3$

5. $S_2 \ \rightarrow_{\texttt{min}} \ S_3$
6. $S_3 \ \rightarrow_{\texttt{min}} \ S_{10} \ f \mid s$
7. $S_3 \ \rightarrow_{\texttt{min}} \ S_{30} \mid t$
8. $S_3 \ \rightarrow_{\texttt{min}} \ S_{20}$

2. Source cell (S_{10})

1. $S_{10} \ f \ \rightarrow_{\texttt{min.min}} \ S_{10} \ s''_k \ (v'_i)\updownarrow_{\forall} \ (f_i)\downarrow_k \mid n''_k \neg v'_k$
2. $S_{10} \ f_k \ \rightarrow_{\texttt{min.min}} \ S_{10} \ (b_i)\updownarrow_k \mid \iota_i$
3. $S_{10} \ a \ s''_k \ n''_k \ \rightarrow_{\texttt{min.min}} \ S_{40} \ r \ d''_k \ (r)\updownarrow$
4. $S_{10} \ b_k \ s''_k \ n''_k \ \rightarrow_{\texttt{min.min}} \ S_{40} \ q \ (q)\updownarrow$
5. $S_{10} \ f \ \rightarrow_{\texttt{min.min}} \ S_{50} \ (g)\updownarrow$

3. Frontier cell (S_{20})

1. $S_{20} \ f_k \ \rightarrow_{\texttt{min.min}} \ S_{20} \ f \ v'' \ s'_k \ (v''_i)\updownarrow_{\forall} \mid \iota_i \ d''_k \neg v''$
2. $S_{20} \ f_j \ \rightarrow_{\texttt{min.min}} \ S_{20} \ f \ v' \ s'_j \ (v'_i)\updownarrow_{\forall} \mid \iota_i \neg v'$
3. $S_{20} \ f_k \ \rightarrow_{\texttt{min.min}} \ S_{20} \ (b_i)\updownarrow_j \mid d''_k \ v'' \ \iota_i$
4. $S_{20} \ f_j \ \rightarrow_{\texttt{min.min}} \ S_{20} \ (b_i)\updownarrow_j \mid v' \ \iota_i$
5. $S_{20} \ b_k \ s''_k \ \rightarrow_{\texttt{min.min}} \ S_{20} \ f$
6. $S_{20} \ f \ \rightarrow_{\texttt{min.min}} \ S_{20} \ v_k \ s''_k \ (f_i)\downarrow_k \mid \iota_i \ v'' \ n''_k \neg v'_k \ v''_k \ d'_k \ d''_k$
7. $S_{20} \ f \ \rightarrow_{\texttt{min.min}} \ S_{20} \ v_k \ s''_k \ (f_i)\downarrow_k \mid \iota_i \ n''_k \neg d \ v'_k \ v''_k \ d'_k \ d''_k$
8. $S_{20} \ f \ \rightarrow_{\texttt{min.min}} \ S_{20} \ v''_k \ s''_k \ (f_i)\uparrow_k \mid \iota_i \ d'_k \neg v''_k$
9. $S_{20} \ f \ s'_j \ \rightarrow_{\texttt{min.min}} \ S_{20} \ (b_i)\downarrow_j \mid d''_j \ \iota_i$
10. $S_{20} \ f \ s'_j \ \rightarrow_{\texttt{min.min}} \ S_{20} \ (b_i)\updownarrow_j \mid \iota_i$

4. Intermediate cell (S_{20}–S_{21})

1. $S_{20} \ g \ \rightarrow_{\texttt{min.min}} \ S_{50} \ (g)\updownarrow$
2. $S_{20} \ v'_k \ \rightarrow_{\texttt{max}} \ S_{20} \mid v'_k$
3. $S_{20} \ v''_k \ \rightarrow_{\texttt{max}} \ S_{20} \mid v''_k$
4. $S_{20} \ a \ s'_j \ s''_k \ \rightarrow_{\texttt{min.min}} \ S_{20} \ d \ d'_j \ d''_k \ (a)\downarrow_j \mid d''_j \neg d'_k$

5. $S_{20}\ a\ s'_j\ s''_k \rightarrow_{\texttt{min.min}} S_{20}\ d\ d'_j\ d''_k\ (a)\updownarrow_j$
6. $S_{20}\ d\ d''_k\ d'_k \rightarrow_{\texttt{min.min}} S_{20}$
7. $S_{20} \rightarrow_{\texttt{min}} S_{21}\ (q)\updownarrow_\forall \mid q$
8. $S_{20} \rightarrow_{\texttt{min}} S_{21}\ w' \mid q\ v' \neg w'$
9. $S_{20} \rightarrow_{\texttt{min}} S_{21}\ w'' \mid q\ v'' \neg w''$
10. $S_{20} \rightarrow_{\texttt{max.min}} S_{21}\ w'_k \mid q\ v'_k \neg w'_k$
11. $S_{20} \rightarrow_{\texttt{max.min}} S_{21}\ w''_k \mid q\ v''_k \neg w''_k$
12. $S_{20} \rightarrow_{\texttt{min}} S_{21}\ (r)\updownarrow_\forall \mid r$
13. $S_{20}\ v' \rightarrow_{\texttt{min}} S_{21} \mid r \neg w'$
14. $S_{20}\ v'' \rightarrow_{\texttt{min}} S_{21} \mid r \neg w''$
15. $S_{20}\ v'_k \rightarrow_{\texttt{max.min}} S_{21} \mid r \neg w'_k$
16. $S_{20}\ v''_k \rightarrow_{\texttt{max.min}} S_{21} \mid r \neg w''_k$
17. $S_{21} \rightarrow_{\texttt{min}} S_{40}$

5. Target cell (S_{30})

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

6. All cells (S_{40}: at the end of each round; S_{50}: at the end of the algorithm.)

1. $S_{40}\ v'_k \rightarrow_{\texttt{max.max}} S_{40} \mid s$
2. $S_{40}\ v''_k \rightarrow_{\texttt{max.max}} S_{40} \mid s$
3. $S_{40}\ v'_k \rightarrow_{\texttt{max.max}} S_{40} \mid t$
4. $S_{40}\ v''_k \rightarrow_{\texttt{max.max}} S_{40} \mid t$
5. $S_{40}\ a \rightarrow_{\texttt{max}} S_{40}$
6. $S_{40}\ q \rightarrow_{\texttt{max}} S_{40}$
7. $S_{40}\ r \rightarrow_{\texttt{max}} S_{40}$
8. $S_{40} \rightarrow_{\texttt{min}} S_3$

9. $S_{50}\ g\ \rightarrow_{\texttt{max}}\ S_{50}$
10. $S_{50}\ d\ \rightarrow_{\texttt{max}}\ S_{50}$
11. $S_{50}\ n'_j\ \rightarrow_{\texttt{max.min}}\ S_{50}$
12. $S_{50}\ n''_k\ \rightarrow_{\texttt{max.min}}\ S_{50}$
13. $S_{50}\ w'_k\ v'_k\ \rightarrow_{\texttt{max.min}}\ S_{50}$
14. $S_{50}\ w''_k\ v''_k\ \rightarrow_{\texttt{max.min}}\ S_{50}$
15. $S_{50}\ w'\ v'\ \rightarrow_{\texttt{max}}\ S_{50}$
16. $S_{50}\ w''\ v''\ \rightarrow_{\texttt{max}}\ S_{50}$

The state-chart of NW-DFS-Node is exactly the same as NW-DFS-Edge (see Figure 5) and most symbols are the same. To simulate the node-splitting technique, NW-DFS-Node replaces symbols of NW-DFS-Edge for cell σ_i as follows:

- $v \rightarrow v'$ and v'', which indicate that σ_i's entry and exit nodes are visited, respectively;
- $w \rightarrow w'$ and w'', which indicate that σ_i's entry and exit nodes are permanently visited, respectively;
- $v_j \rightarrow v'_j$ and v''_j, which indicate that σ_j's entry and exit nodes are visited, respectively.
- $w_j \rightarrow w'_j$ and w''_j, which indicate that σ_j's entry and exit nodes are permanently visited, respectively.

Differently from the edge-disjoint algorithm, an unvisited intermediate cell, σ_i, becomes a frontier cell by accepting (1) a push-back visit token, indicated by f_k and d''_k, if its exit node is unvisited, indicated by v'' (rule 3.2) or (2) a forward visit token, f_j, if its entry node is unvisited, indicated by v' (rule 3.1). In both cases, σ_i sends its visiting notification, v''_i, respectively v'_i.

As mentioned in the explanations of Figure 4, consider a frontier cell, σ_i, indicated by f. If σ_i is currently visited on its exit node, indicated by v'', case (Ia) is allowed—σ_i can continue with a forward search (rule 3.6). Otherwise, if σ_i is currently visited on its entry node and it appears in a disjoint path, indicated by d, case (IIa) is not allowed, to constrain its in-flow to one (rule 3.7). To conform to the earlier mentioned visiting restriction, we avoid visiting a cell on its entry node, which has been first visited on its exit node, using inhibitor v''_k (rules 3.6–7). If cell σ_i is visited on its exit node, this must be the last visiting on σ_i in the same round (because a cell can not be first visited on its exit node and later on its entry node). Thus, for case (IIc), to send a backtrack token, if cell σ_i has a sp-predecessor of its exit node, indicated by s'_j and d''_j, σ_i sends the backtrack token to σ_j (rule 3.9).

Consider an intermediate cell, σ_i, which has been visited on both its entry and exit nodes. In such case, it has two sp-predecessors and two sp-successors. When σ_i receives

a path confirmation, a, for the first time, σ_i relays it to the sp-predecessor of its exit node, indicated by s'_j and d''_j, and transforms the sp-successor of its exit node, indicated by s''_j and inhibitor d'_k (not the sp-successor of its entry node) to a dp-successor (rule 4.4). Later when σ_i receives a path confirmation, for the second time, there are only one sp-predecessor and one sp-successor to relay it (rule 4.5).

In **Figure 6**, the search path, $\sigma_0.\sigma_2.\sigma_3.\sigma_2.\sigma_1.\sigma_6.\sigma_7.\sigma_3.\sigma_2.\sigma_4.\sigma_5.\sigma_{10}$, has visited cell σ_2 twice—once by a forward search to its entry node (rules 3.2, 3.7) and again by a push-back on its exit node (rule 3.1). At this time, σ_2 contains $\{d'_1, d''_3, s'_5, s''_1, s'_3, s''_6\}$. When σ_2 receives a path confirmation, a, for the first time, σ_2 relays it to the sp-predecessor of its exit node, σ_3, indicated by s'_3 and d''_3, and transforms the corresponding sp-successor, indicated by s''_6 and inhibitor d'_1 (s''_1 should not be transformed) to the dp-successor, d''_6.

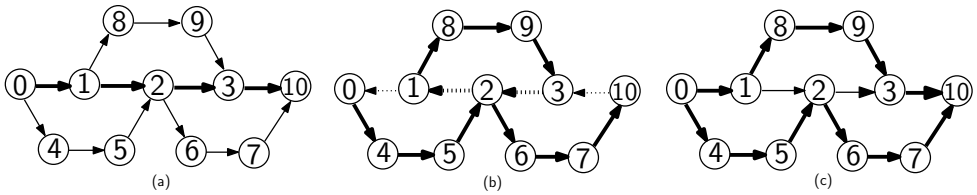


Figure 6: An example for node-disjoint paths.

The following proposition sums up the above arguments:

Proposition 8. When P Algorithm 2 ends, disjoint path predecessors and successors symbols in its output indicate a maximal cardinality set of node-disjoint paths.

6 Our BFS-based Algorithms

Our BFS-based algorithms also work in successive search rounds.

Each search round starts with a set of disjoint paths, which is empty at the start of the first round. The source cell, σ_s, starts a search wave, to find new augmenting paths. Current "frontier"cells send out visit tokens. The cells which receive and accept these visit tokens become the new frontier, by inserting themselves at the front of current search paths. The advancing wave periodically sends *progress indicators* back to the source: (a) *extending notifications* (at least one search path is still extending) and (b) *path confirmations* (at least one search path was successful, i.e. a new augmenting path was found). While moving towards the source, each path confirmation reshapes the existing paths and the newly found augmenting path, by discarding cancelling arcs and relinking the rest, building a larger set of disjoint paths.

If no progress indicator arrives in the expected time, σ_s assumes that the search round ends. If at least one search path was successful (at least one augmenting path was confirmed), σ_s initiates a global broadcast, carried by a reset token, which reinitialises all cells for the next search round. Otherwise, σ_s initiates a global broadcast, carried by a finalise token, informing all cells that the algorithm has terminated.

In each search round, an intermediate cell, σ_i, can be visited only once. Several search paths may try to visit the same intermediate cell *simultaneously*, but only one of them

succeeds. **Figure 7 (a)** shows such a scenario: cells σ_1, σ_2 and σ_3 try to visit cell σ_4, in the same step; but only cell σ_1 succeeds, via arc (σ_1, σ_4). We solve this *choice* problem by using the `min.min` mode; such a `min.min` rule is instantiated only once, selecting one of the received visit tokens and ignoring the others (see rules for 3. frontier cell).

The target cell, σ_t, faces an additional decision problem. When several search paths arrive, simultaneously or sequentially, σ_t must quickly decide which search paths can succeed and which ones must be ignored (in the same round). We solve this problem using a *branch-cut* strategy, based on branch IDs [14]. Given a search path, $\tau = \sigma_s.\sigma_{sr}.\tau'$, its *branch ID* is the cell ID of its *first intermediate* cell after the source taken by τ, i.e. r. A frontier cell, σ_i, forwards the branch ID on top of its visit token, e.g, $f_{r,i}$.

The following result is straightforward:

Proposition 9. In any search round, search paths which share the same branch ID are incompatible; only one of them can succeed and become an augmenting path.

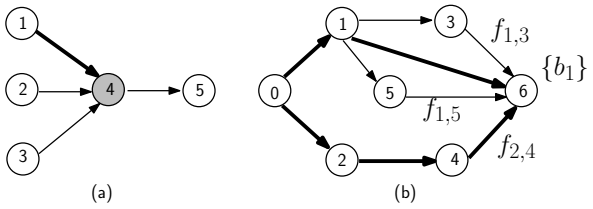


Figure 7: BFS challenges: (a) A choice made between several search paths visiting the same cell, (b) Search paths sharing the same branch ID are incompatible.

Figure 7 (b) shows four search paths arriving at cell σ_6: $\pi = \sigma_0.\sigma_1.\sigma_6$, $\tau_1 = \sigma_0.\sigma_1.\sigma_3.\sigma_6$, $\tau_2 = \sigma_0.\sigma_1.\sigma_5.\sigma_6$ and $\tau_3 = \sigma_0.\sigma_2.\sigma_4.\sigma_6$; their branch IDs are 1, 1, 1 and 2, respectively. Assume that π arrives first and succeeds as an augmenting path; in this case, σ_t records π's branch ID, 1. Consider the fate of the other search paths, τ_1, τ_2 and τ_3, which attempt to reach the target 6, later in the same round. τ_1 and τ_2 fail, because they share π's recorded branch ID, 1. However, τ_3 succeeds as a new augmenting path, because it has a different branch ID, 2.

A branch ID, r, is recorded via symbol b_r, in the target cell, σ_t. Technically, it is important that recording symbols b_r are used as promoters. which enable rules, without being consumed. Otherwise, objects b_r can be consumed before completing their role; e.g., in **Figure 7 (b)**, the rejection of τ_1 would consume b_1, and there would be nothing left to reject τ_2.

6.1 Rules for BFS-based Edge-disjoint Paths

The P system rules for edge- and node-disjoint paths are slightly different, due to the simulated node-splitting approach, but the basic principle is the same.

P Algorithm 3: NW-BFS-Edge

Input: As in the edge-disjoint paths algorithm of P Algorithm 1.

Output: As in the edge-disjoint paths algorithm of P Algorithm 1.

Table 5 shows the initial and final configurations of the P system defined by P Algorithm 3, for Figure 1.

Table 5: Initial and final configurations of P Algorithm 3, 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	$S_{50}\ \iota_0\ s\ d''_1\ d''_2\ d''_3$	$S_{50}\ \iota_1\ d'_0\ d''_4$	$S_{50}\ \iota_2\ d'_0\ d''_6$	$S_{50}\ \iota_3\ d'_0\ d''_5$
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	$S_{50}\ \iota_4\ d'_1\ d''_7$	$S_{50}\ \iota_5\ d'_3\ d''_7$	$S_{50}\ \iota_6\ d'_2\ d''_7$	$S_{50}\ \iota_7\ t\ d'_4\ d'_5\ d'_6$

1. Initial differentiation (S_1–S_3)

1. $S_0\ n\ \rightarrow_{\texttt{min.min}}\ S_1\ (n)\updownarrow_\forall$
2. $S_0\ \rightarrow_{\texttt{min}}\ S_0\ n \mid s$
3. $S_1\ \rightarrow_{\texttt{min}}\ S_2$
4. $S_2\ n\ \rightarrow_{\texttt{max}}\ S_3$
5. $S_2\ \rightarrow_{\texttt{min}}\ S_3$
6. $S_3\ \rightarrow_{\texttt{min}}\ S_{10}\ f \mid s$
7. $S_3\ \rightarrow_{\texttt{min}}\ S_{30} \mid t$
8. $S_3\ \rightarrow_{\texttt{min}}\ S_{20}$

2. Source cell (S_{10})

1. $S_{10}\ f\ \rightarrow_{\texttt{min}}\ S_{10}\ x\ (f_i)\downarrow\ \mid \iota_i$
2. $S_{10}\ x\ y\ a_k\ \rightarrow_{\texttt{min.min}}\ S_{10}\ d''_k\ x\ d$
3. $S_{10}\ x\ y\ e_k\ \rightarrow_{\texttt{min}}\ S_{10}\ x$
4. $S_{10}\ x\ y\ \rightarrow_{\texttt{min}}\ S_{40}\ (q)\updownarrow_\forall\ \mid o$
5. $S_{10}\ x\ y\ \rightarrow_{\texttt{min}}\ S_{50}\ (g)\updownarrow_\forall$
6. $S_{10}\ x\ \rightarrow_{\texttt{min}}\ S_{10}\ x\ y$
7. $S_{10}\ a_k\ \rightarrow_{\texttt{min.min}}\ S_{10}\ d''_k\ o$
8. $S_{10}\ e_k\ \rightarrow_{\texttt{max}}\ S_{10}$
9. $S_{10}\ f_{k,j}\ \rightarrow_{\texttt{max.min}}\ S_{10}$
10. $S_{10}\ p_{k,j}\ \rightarrow_{\texttt{max.min}}\ S_{10}$

3. Frontier cell (S_{20})

1. $S_{20}\ f_j\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v\ s'_j\ (p_{i,i})\uparrow\ (f_{i,i})\downarrow\ (e_i)\uparrow_j\ \mid \iota_i\ \neg\ v\ d'_j$
2. $S_{20}\ f_{k,j}\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v\ s'_j\ (p_{k,i})\uparrow\ (f_{k,i})\downarrow\ (e_i)\uparrow_j\ \mid \iota_i\ \neg\ v\ d'_j$
3. $S_{20}\ p_{k,j}\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v\ s'_j\ (p_{k,i})\uparrow\ (f_{k,i})\downarrow\ (e_i)\uparrow_j\ \mid \iota_i\ d''_j\ \neg\ v$

4. Intermediate cell (S_{20})

1. $S_{20}\ \rightarrow_{\texttt{min.min}}\ S_{40}\ (q)\updownarrow_\forall\ \mid q$
2. $S_{20}\ \rightarrow_{\texttt{min.min}}\ S_{50}\ (g)\updownarrow_\forall\ \mid g$
3. $S_{20}\ e_k\ \rightarrow_{\texttt{min.min}}\ S_{20}\ (e_i)\updownarrow_j\ \mid \iota_i\ s'_j$
4. $S_{20}\ e_k\ \rightarrow_{\texttt{max.min}}\ S_{20}$
5. $S_{20}\ a_k\ s'_j\ \rightarrow_{\texttt{min.min}}\ S_{20}\ d'_j\ d''_k\ (a_i)\updownarrow_j\ \mid \iota_i$
6. $S_{20}\ d'_j\ d''_j\ \rightarrow_{\texttt{min.min}}\ S_{20}$
7. $S_{20}\ f_{k,j}\ \rightarrow_{\texttt{max.min}}\ S_{20}$
8. $S_{20}\ p_{k,j}\ \rightarrow_{\texttt{max.min}}\ S_{20}$
9. $S_{20}\ f_j\ \rightarrow_{\texttt{max.min}}\ S_{20}$

5. Target cell (S_{30})

1. $S_{30}\ \rightarrow_{\texttt{min.min}}\ S_{40}\ (q)\updownarrow\ \mid q$
2. $S_{30}\ \rightarrow_{\texttt{min.min}}\ S_{50}\ (g)\updownarrow\ \mid g$
3. $S_{30}\ f_j\ \rightarrow_{\texttt{max.min}}\ S_{30}\ \mid d'_j$
4. $S_{30}\ f_j\ \rightarrow_{\texttt{max.min}}\ S_{30}\ d'_j\ (a_i)\updownarrow_j\ \mid \iota_i\ \neg\ d'_j$
5. $S_{30}\ f_{k,j}\ \rightarrow_{\texttt{max.min}}\ S_{30}\ \mid d'_j$
6. $S_{30}\ f_{k,j}\ \rightarrow_{\texttt{max.min}}\ S_{30}\ \mid b_k$
7. $S_{30}\ f_{k,j}\ \rightarrow_{\texttt{max.min}}\ S_{30}\ d'_j\ b_k\ (a_i)\updownarrow_j\ \mid \iota_i\ \neg\ d'_j\ b_k$

6. All cells (S_{40}–S_{41}: at the end of each round; S_{50}: at the end of the algorithm.)

1. $S_{40}\ f_{k,j}\ \rightarrow_{\texttt{max.min}}\ S_{41}$
2. $S_{40}\ p_{k,j}\ \rightarrow_{\texttt{max.min}}\ S_{41}$
3. $S_{40}\ f_j\ \rightarrow_{\texttt{max.min}}\ S_{41}$

4. $S_{40}\ o\ \rightarrow_{\texttt{max}}\ S_{41}$
5. $S_{40}\ v\ \rightarrow_{\texttt{max}}\ S_{41}$
6. $S_{40}\ q\ \rightarrow_{\texttt{max}}\ S_{41}$
7. $S_{40}\ s'_j\ \rightarrow_{\texttt{max.min}}\ S_{41}$
8. $S_{41}\ q\ \rightarrow_{\texttt{max}}\ S_3$
9. $S_{41}\ \rightarrow_{\texttt{min}}\ S_3$
10. $S_{50}\ g\ \rightarrow_{\texttt{max}}\ S_{50}$
11. $S_{50}\ b_j\ \rightarrow_{\texttt{max.min}}\ S_{50}$

Figure 5 shows the state transitions of P Algorithm 3.

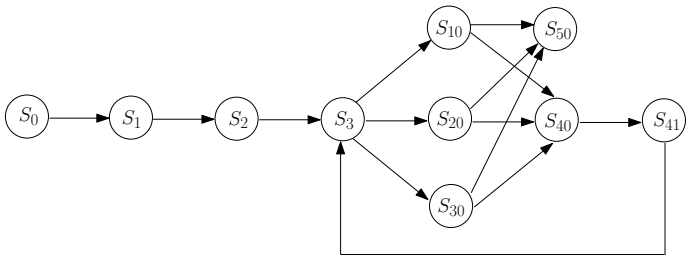


Figure 8: State-chart of our BFS-based algorithms.

This algorithm has two phases: (1) Phase I (rules for initial differentiation), which prompts each cell to enter its corresponding state (neighbourhood knowledge is not needed in the BFS-based solution):

- Source cell, σ_s: S_{10}
- Target cell, σ_t: S_{30}
- Intermediate cells, σ_i: S_{20}

(2) Phase II (rules specifically for the source cell, frontier cells, intermediate cells and the target cell, and common rules for all cells), which searches disjoint paths.

In Phase I, the source cell, σ_s generates one symbol, n. This simulates that σ_s receives a symbol n from a non-existing cell (rule 1.2). On receiving one or more n's, each cell, σ_i, discards one n and broadcasts n, to all its neighbours (rule 1.1), Any superfluous n's are discarded after one step (rule 1.4). Next, each cell enters its corresponding state (rules 1.6–8).

In Phase II, cell σ_i uses the following symbols to record its relationships with cell σ_j: d'_j indicates a dp-predecessor; d''_k indicates a dp-successor; s'_j indicates a sp-predecessor; b_k records the branch ID of an augmenting path, specifically for the target cell, σ_t.

The following symbols are sent by cell σ_i: f_s is the forward visit token, sent by the source cell, σ_s; $f_{k,i}$ is the forward visit token, where k is the branch ID; $p_{k,i}$ is the push-back visit token, where k is the branch ID; e_i is the extending notification; a_i is the path confirmation; q is the reset token; g is the finalise token.

Specifically, the source cell, σ_s uses the following symbols to record its state: o indicates that an augmenting path was found in the current round; x, y implement a timer to wait for progress indicators. An intermediate cell, σ_i, uses d to indicate that it appears in a disjoint path.

For example, **Figure 1 (c)** shows a disjoint path, $\sigma_0.\sigma_2.\sigma_5.\sigma_7$, and an augmenting path, $\sigma_0.\sigma_3.\sigma_5.\sigma_2.\sigma_6.\sigma_7$. **Table 6** shows cells' contents at stage (c). Cell σ_5 contains: a dp-predecessor, d'_2, a dp-successor, d''_7, a sp-predecessor, s'_3. Cell σ_2 contains: a dp-predecessor, d'_0, a dp-successor, d''_5, a sp-predecessor, s'_5.

The evolution of the source cell: In each round, the source cell, σ_s, starts the search wave by sending forward visit tokens, f_s, to all its children (rule 2.1). Then, σ_s waits one step for the progress indicators (rule 2.2 for path confirmations, rule 2.3 for extending notifications). If no expected progress indicator arrives, then σ_s signals the end of the round. If a new augmenting path was found, indicated by o, σ_s initiates a global broadcast, carried by a reset token, q (rule 2.4). Otherwise, σ_s initiates a global broadcast, carried by a finalise token, g (rule 2.5).

The evolution of frontier cells: An unvisited intermediate cell, σ_i, becomes a frontier cell by accepting either (1) a forward visit token from σ_s, f_s (rule 3.1), or (2) a forward visit token, $f_{k,j}$ (rule 3.2), or a push-back visit token, $p_{k,j}$ (rule 3.3), from another frontier cell, where k is the branch ID. In case (1), σ_i is the first intermediate cell on the current search path, so it sends forward visit tokens, $f_{i,i}$, to its children, and push-back visit token, $p_{i,i}$, to its parents (rule 3.1). In case (2), σ_i sends out forward visit tokens, $f_{k,i}$, to its children, and push-back visit tokens, $p_{k,i}$ to its parents (rules 3.2, 3.3). In both cases (1) and (2), σ_i records a sp-predecessor, s'_s, respectively s'_j, and sends back an extending notification, e_s, respectively e_i. When σ_i sends its visit tokens, the frontier advances.

The evolution of other intermediate cells: A previous frontier cell, σ_i, relays progress indicators: extending notifications, e_k (rule 4.3) and path confirmations, a_k (rule 4.5). On receiving path confirmations, a_k, σ_i transforms its sp-predecessor, s'_j, into a dp-predecessor, d'_j, and records its dp-successor, d''_k (rule 4.5). Then, each end of the cancelling arc pair, d'_j and d''_j, are deleted, if any (rule 4.6).

The evolution of the target cell: The target cell, σ_t, accepts either (1) a forward visit token from σ_s, f_s (rule 5.4), or (2) a forward visit token from a frontier cell σ_k, $f_{k,j}$, where k is the branch ID, recorded as b_k (rule 5.7). In both cases (1) and (2), σ_t records a dp-predecessor, d'_s, respectively d'_j, and sends back a path confirmation, a_t.

Table 6 shows P Algorithm 3 trace fragments for stages (a), (c) and (e) of Figure 1. Symbols, d'_j and d''_k, indicate dp-predecessors and dp-successors, respectively; symbols s'_j indicate sp-predecessors. In stage 1(a), the two disjoint paths, $\sigma_0.\sigma_1.\sigma_4.\sigma_7$ and $\sigma_0.\sigma_2.\sigma_5.\sigma_7$, are recorded by the following cell contents: $\sigma_0 : \{d''_1, d''_2\}$, $\sigma_1 : \{d'_0, d''_4\}$, $\sigma_2 : \{d'_0, d''_5\}$, $\sigma_4 : \{d'_1, d''_7\}$, $\sigma_5 : \{d'_2, d''_7\}$, $\sigma_7 : \{d'_4, d''_5\}$. In stage 1(c), the successful search path $\sigma_0.\sigma_3.\sigma_5.\sigma_2.\sigma_6.\sigma_7$ is recorded as: $\sigma_3 : \{s'_0\}$, $\sigma_5 : \{s'_3\}$, $\sigma_2 : \{s'_5\}$, $\sigma_6 : \{s'_2\}$, $\sigma_7 : \{d'_6\}$ (the target records d'_6 directly). In stage 1(e), there are three disjoint paths, $\sigma_0.\sigma_1.\sigma_4.\sigma_7$,

$\sigma_0.\sigma_2.\sigma_6.\sigma_7$ and $\sigma_0.\sigma_3.\sigma_5.\sigma_7$, which are recorded as: $\sigma_0 : \{d''_1, d''_2, d''_3\}$, $\sigma_1 : \{d'_0, d''_4\}$, $\sigma_2 : \{d'_0, d''_6\}$, $\sigma_3 : \{d'_0, d''_5\}$, $\sigma_4 : \{d'_1, d''_7\}$, $\sigma_5 : \{d'_3, d''_7\}$, $\sigma_6 : \{d'_2, d''_7\}$, $\sigma_7 : \{d'_4, d'_5, d'_6\}$.

Table 6: P Algorithm 3 trace fragments for stages (a), (c) and (e) of Figure 1.

Stage	σ_0	σ_1	σ_2	σ_3
1(a)	$S_2\ \iota_0\ s\ d''_1\ d''_2\ \ldots$	$S_2\ \iota_1\ d'_0\ d''_4\ \ldots$	$S_2\ \iota_2\ d'_0\ d''_5\ \ldots$	$S_2\ \iota_3\ \ldots$
1(c)	$S_2\ \iota_0\ s\ d''_1\ d''_2\ \ldots$	$S_0\ \iota_1\ d'_0\ d''_4\ \ldots$	$S_2\ \iota_2\ d'_0\ d''_5\ s'_5\ \ldots$	$S_2\ \iota_3\ s'_0\ \ldots$
1(e)	$S_3\ \iota_0\ s\ d''_1\ d''_2\ d''_3$	$S_3\ \iota_1\ d'_0\ d''_4$	$S_3\ \iota_2\ d'_0\ d''_6$	$S_3\ \iota_3\ d'_0\ d''_5$
Stage	σ_4	σ_5	σ_6	σ_7
1(a)	$S_2\ \iota_4\ d'_1\ d''_7\ \ldots$	$S_2\ \iota_5\ d'_2\ d''_7\ \ldots$	$S_2\ \iota_6\ \ldots$	$S_4\ \iota_7\ t\ d'_4\ d'_5\ \ldots$
1(c)	$S_0\ \iota_4\ d'_1\ d''_7\ \ldots$	$S_2\ \iota_5\ d'_2\ d''_7\ s'_3\ \ldots$	$S_2\ \iota_6\ s'_2\ \ldots$	$S_4\ \iota_7\ t\ d'_4\ d'_5\ d'_6\ \ldots$
1(e)	$S_3\ \iota_4\ d'_1\ d''_7$	$S_3\ \iota_5\ d'_3\ d''_7$	$S_3\ \iota_6\ d'_2\ d''_7$	$S_3\ \iota_7\ t\ d'_4\ d'_5\ d'_6$

The preceding arguments indicate a bisimulation relation between our BFS-based algorithm and the classical Edmonds-Karp BFS-based algorithm for edge-disjoint paths [6]. The following proposition encapsulates these arguments:

Proposition 10. When P Algorithm 3 terminates, disjoint path predecessor and successor symbols indicated in its output section indicate a maximal cardinality set of edge-disjoint paths.

6.2 Rules for BFS-based Node-disjoint Paths

P Algorithm 4: NW-BFS-Node

Input: As in the edge-disjoint paths algorithm of P Algorithm 1.

Output: Similar to the edge-disjoint paths algorithm. However, the predecessor and successor symbols indicate *node-disjoint paths*, instead of edge-disjoint paths.

Table 7 shows the initial and final configurations of the P system defined by P Algorithm 4, for Figure 6.

Table 7: Initial and final configurations of P Algorithm 4, for Figure 6.

Cell	σ_0	σ_1	σ_2	σ_3	σ_4	σ_5
Initial	$S_0\ \iota_0\ s$	$S_0\ \iota_1$	$S_0\ \iota_2$	$S_0\ \iota_3$	$S_0\ \iota_4$	$S_0\ \iota_5$
Final	$S_3\ \iota_0\ s\ d''_1\ d''_4$	$S_3\ \iota_1\ d'_0\ d''_8$	$S_3\ \iota_2\ d'_5\ d''_6$	$S_3\ \iota_3\ d'_9\ d''_{10}$	$S_3\ \iota_4\ d'_0\ d''_5$	$S_3\ \iota_5\ d'_4\ d''_2$
Cell	σ_6	σ_7	σ_8	σ_9	σ_{10}	
Initial	$S_0\ \iota_6$	$S_0\ \iota_7$	$S_0\ \iota_8$	$S_0\ \iota_9$	$S_0\ \iota_{10}\ t$	
Final	$S_3\ \iota_6\ d'_2\ d''_7$	$S_3\ \iota_7\ d'_6\ d''_{10}$	$S_3\ \iota_8\ d'_1\ d''_9$	$S_3\ \iota_9\ d'_8\ d''_3$	$S_3\ \iota_{10}\ t\ d'_3\ d'_7$	

The rules of P Algorithm 4 are the same as the rules of P Algorithm 3, except that the rules for frontier and intermediate cells are replaced by the following groups of rules.

3. Frontier cell (S_{20})

1. $S_{20}\ f_j\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v'\ s'_j\ (p_{i,i})\uparrow\ (f_{i,i})\downarrow\ (e_i)\uparrow_j\ \mid \iota_i \neg\ v'\ d'_j$
2. $S_{20}\ f_{k,j}\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v'\ s'_j\ (p_{k,i})\uparrow\ (e_i)\uparrow_j\ \mid \iota_i\ d \neg\ v'\ d'_j$

3. $S_{20}\ f_{k,j}\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v'\ s'_j\ (p_{k,i})\uparrow\ (f_{k,i})\downarrow\ (e_i)\uparrow_j\ \mid \iota_i\ \neg\ d\ v'\ d'_j$

4. $S_{20}\ p_{k,j}\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v''\ s'_j\ (p_{k,i})\uparrow\ (f_{k,i})\downarrow\ (e_i)\downarrow_j\ \mid \iota_i\ d''_j\ \neg\ v''$

4. Intermediate cell (S_{20})

1. $S_{20}\ \rightarrow_{\texttt{min.min}}\ S_{40}\ (q)\updownarrow_\forall\ \mid q$

2. $S_{20}\ \rightarrow_{\texttt{min.min}}\ S_{50}\ (g)\updownarrow_\forall\ \mid g$

3. $S_{20}\ e_k\ \rightarrow_{\texttt{min.min}}\ S_{20}\ (e_i)\uparrow_l\ \mid \iota_i\ s'_j\ d''_j\ s'_l\ d'_k$

4. $S_{20}\ e_k\ \rightarrow_{\texttt{min.min}}\ S_{20}\ (e_i)\downarrow_j\ \mid \iota_i\ s'_j\ d''_j$

5. $S_{20}\ e_k\ \rightarrow_{\texttt{min.min}}\ S_{20}\ (e_i)\updownarrow_j\ \mid \iota_i\ s'_j$

6. $S_{20}\ e_k \rightarrow_{\texttt{max.min}}\ S_{20}$

7. $S_{20}\ a_k\ s'_j\ \rightarrow_{\texttt{min.min}}\ S_{20}\ d\ d'_j\ d''_k\ (a_i)\downarrow_j\ \mid \iota_i\ d''_j$

8. $S_{20}\ a_k\ s'_j\ \rightarrow_{\texttt{min.min}}\ S_{20}\ d\ d'_j\ d''_k\ (a_i)\updownarrow_j\ \mid \iota_i$

9. $S_{20}\ d'_j\ d''_j\ d\ \rightarrow_{\texttt{min.min}}\ S_{20}$

10. $S_{20}\ f_{k,j}\ \rightarrow_{\texttt{max.min}}\ S_{20}$

11. $S_{20}\ p_{k,j}\ \rightarrow_{\texttt{max.min}}\ S_{20}$

12. $S_{20}\ f_j\ \rightarrow_{\texttt{max.min}}\ S_{20}$

To simulate the node-splitting technique, cell σ_i uses v' and v'' to indicate its visited entry and exit nodes, respectively.

Differently from the edge-disjoint algorithm, an unvisited intermediate cell, σ_i, becomes a frontier cell by accepting a forward visit token, f_j or $f_{k,j}$, if its entry node is unvisited, indicated by v' (rules 3.1–3) or by accepting a push-back visit token, $p_{k,j}$, if it exit node is unvisited, indicated by v'' (rule 3.4). To constrain the in-flow to one, on accepting a forward visit token, cell σ_i can only send push-back visit tokens, if it appears in a disjoint path, indicated by d (rule 3.2); otherwise, it sends both forward and push-back tokens (rule 3.3).

Our BFS-based solution confirms to the earlier mentioned visiting restriction. When cell σ_i is visited on its exit node, it immediately visits its entry node via the internal arc, by sending push-back visit tokens (rule 3.4), so no loop occurs.

Consider an intermediate cell, σ_i, which has been visited on both its entry and exit nodes. It must be first visited on its entry node and later on its exit node. In such case, it has two sp-predecessors. We implement following rules for σ_i to relay two path confirmations and periodical extending notifications.

When σ_i receives a path confirmation, a_k, for the first time, σ_i relays it to the sp-predecessor of its exit node, indicated by s'_j and d''_j, and transforms s'_j into d'_j and records

d''_k (rule 4.7). Later when σ_i receives a path confirmation, for the second time, there is only one sp-predecessor to relay it to (rule 4.5).

The extending notifications, e_k's, are received periodically. If e_k comes from the sp-successor of it entry node, i.e. its dp-predecessor, indicated by d'_k, σ_i should relay it to the sp-predecessor of it entry node, indicated by s'_l, rather than the sp-predecessor of its exit node, indicated by s'_j and d''_j (rule 4.3). Otherwise, if e_k is not from the sp-successor of its entry node, σ_i relays it to the sp-predecessor of its exit node, indicated by s'_j and d''_j (rule 4.4).

In **Figure 6**, the search path, $\sigma_0.\sigma_2.\sigma_3.\sigma_2.\sigma_1.\sigma_6.\sigma_7.\sigma_3.\sigma_2.\sigma_4.\sigma_5.\sigma_{10}$, has visited cell σ_2 twice—once by a forward search on its entry node (rules 3.2) and again by a push-back on its exit node (rule 3.4). At this time, σ_2 contains $\{d'_1, d''_3, s'_5, s'_3\}$. When σ_2 receives e_1 from the sp-successor of its entry node, σ_1, indicated by d'_1, σ_2 relays it to the sp-predecessor of its entry node, σ_5, indicated by s'_5, rather than the sp-predecessor of its exit node, σ_3, indicated by s'_3 and d''_3 (rule 4.3). When σ_2 receives e_6 (rule 4.4) or a path confirmation, a_6 (rule 4.7), from σ_6, σ_2 relays it to σ_3, indicated by s'_3 and d''_3.

The following proposition sums up all these arguments:

Proposition 11. When P Algorithm 4 ends, disjoint path predecessors and successors symbols in its output indicate a maximal cardinality set of node-disjoint paths.

7 Runtime Performance of P Systems Solutions

Consider a simple P system with n cells and m arcs, where f_e = the maximum number of edge-disjoint paths, f_n = the maximum number of node-disjoint paths and d = the outdegree of the source cell. Dinneen et al. showed that their *graph*-based edge- and node-disjoint algorithms run in $O(mn)$ P steps [4]. Their modified *digraph*-based versions, used in this article, DKN-DFS-Edge and DKN-DFS-Node, share the same upper-bound. However, a closer inspection, not detailed here, shows that this upper-bound can be improved.

Proposition 12. Dinneen et al.'s DKN-DFS-Edge and DKN-DFS-Node algorithms run in $O(md)$ P steps.

In our NW-DFS-Edge and NW-DFS-Node algorithms, the source cell starts d search rounds. In each search round, for NW-DFS-Edge, previously visited cells are not revisited, so the search path visits at most n cells; while for NW-DFS-Node, each intermediate cell can be visited at most twice (once via the virtual entry node and again via the virtual exit node), so the search path visits at most $2n - 2$ cells. As earlier mentioned, the housekeeping operations required to avoid unnecessary visits are performed in parallel with the main search. Thus, NW-DFS-Edge and NW-DFS-Node run in $O(nd)$ P steps, which is asymptotically faster than the $O(md)$ for DKN-DFS-Edge and DKN-DFS-Node.

Proposition 13. Our NW-DFS-Edge and NW-DFS-Node algorithms run in $O(nd)$ P steps.

In our NW-BFS-Edge and NW-BFS-Node algorithms, at least one augmenting path is found in each search round, so the number of search rounds is at most f_e or f_n,

respectively. In each round, for NW-BFS-Edge, each cell can be visited only once, so the search path visits at most n cells; while for NW-BFS-Node, each intermediate cell can be visited at most twice, so the search path visits at most $2n-2$ cells. Thus, NW-BFS-Edge and NW-BFS-Node run in $O(nf_e)$ and $O(nf_n)$ P steps, respectively. As $f_e, f_n \leq d$, this is asymptotically faster than the $O(nd)$ for NW-DFS-Edge and NW-DFS-Node.

Proposition 14. Our NW-BFS-Edge and NW-BFS-Node algorithms run in $O(nf)$ P steps, where $f = f_e$, for NW-BFS-Edge, and $f = f_n$, for NW-BFS-Node.

Table 8 compares the asymptotic complexity of our algorithms against other published algorithms. All our algorithms score well, in their category, because they leverage the potentially unbounded parallelism inherent in P systems.

Table 8: Asymptotic worst-case complexity comparison.

Algorithm	Runtime Complexity
Ford-Fulkerson (DFS, restricted to unit capacity)	$O(mf)$ sequential steps
Edmonds-Karp (BFS, restricted to unit capacity)	$O(mn)$ sequential steps
DKN-DFS-Edge and -Node	$O(md)$ P steps
NW-DFS-Edge and -Node	$O(nd)$ P steps
NW-BFS-Edge and -Node	$O(nf)$ P steps

Proposition 14 indicates a worst-case. A typical search path does not visit all n cells. Also, BFS-based algorithms frequently find several augmenting paths in the same round, thus the number of rounds is smaller than f. Therefore, the expected runtime is probably much less than the upper bound indicated by Proposition 14.

In order to see the performance improvement using Cidon's distributed DFS techniques and our new result, Theorem 6, we compare various versions, which are modifications based on our NW-DFS-Edge algorithm (P Algorithm 1):

- NW-DFS-Edge /Cidon- /P6-, which uses neither Cidon's distributed DFS techniques nor Theorem 6;
- NW-DFS-Edge /Cidon+ /P6-, which uses Cidon's distributed DFS techniques but not Theorem 6;
- NW-DFS-Edge /Cidon+ /P6+ (our NW-DFS-Edge algorithm), which uses both Cidon's distributed DFS techniques and Theorem 6.

We experimentally compare the performance of these algorithms with DKN-DFS-Edge and our NW-BFS-Edge algorithms, on digraphs randomly generated using NetworkX [9]. We test these algorithms on digraphs with n cells and m arcs, where $n \in \{100, 200, 300, 400\}$ and $m \in \{250, 500, 1000, 1500, 2000, 2500, 3000\}$.

Figures 9, 10, 11 and 12 show the results for cases $n = 100, 200, 300, 400$, respectively, where each plotted result is the average performance (P steps) of thirty digraphs with the same n cells and m arcs. **Tables 9, 10, 11 and 12** show the speed up percentages of the plotted results (average performance) in Figures 9, 10, 11 and 12, for NW-DFS-Edge and NW-BFS-Edge over DKN-DFS-Edge, respectively.

We can see that NW-BFS-Edge algorithm is the fastest, NW-DFS-Edge comes next, and DKN-DFS-Edge is the slowest. Both Cidon's distributed DFS strategies and our new result, Theorem 6, improve the performance, and Cidon's DFS strategies improves more. We compute the average of speed up percentages in Tables 9, 10 and 11, for NW-BFS-Edge and NW-DFS-Edge. Overall, our NW-BFS-Edge algorithm is 39% faster than DKN-DFS-Edge and our NW-BFS-Edge algorithm is 80% faster than DKN-DFS-Edge.

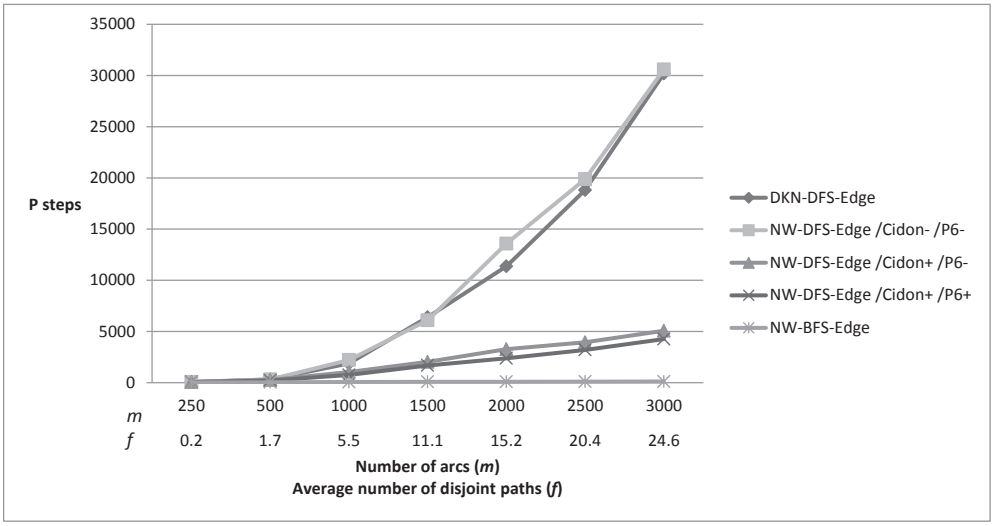


Figure 9: Average performance (P steps) of DKN-DFS-Edge, NW-DFS-Edge /Cidon-/P6-, NW-DFS-Edge /Cidon+ /P6-, NW-DFS-Edge /Cidon+ /P6+ and NW-BFS-Edge algorithms, of thirty digraphs with the same $n = 100$ cells and m arcs. f shows the average number of edge-disjoint paths of thirty digraphs with the same n cells and m arcs.

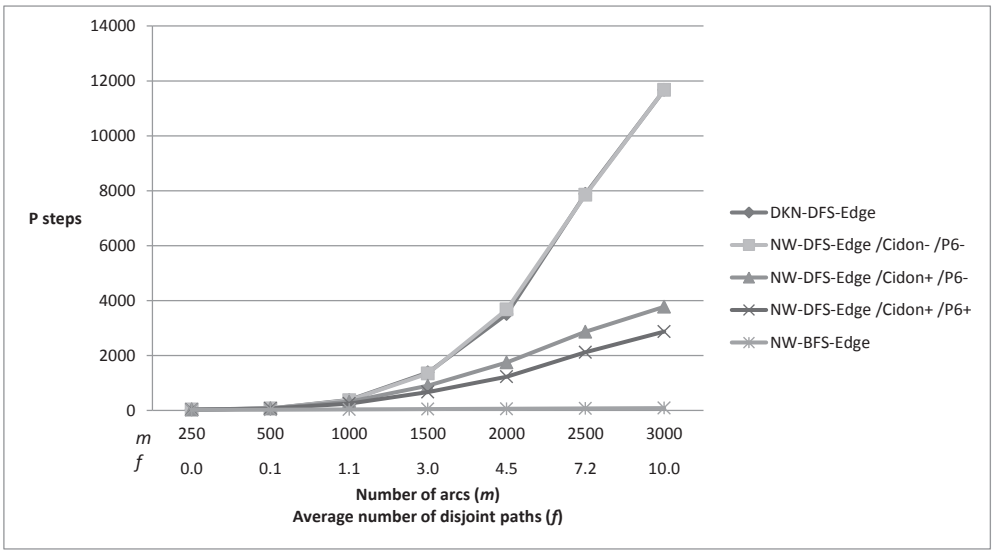


Figure 10: Average performance (P steps) of DKN-DFS-Edge, NW-DFS-Edge /Cidon- /P6-, NW-DFS-Edge /Cidon+ /P6-, NW-DFS-Edge /Cidon+ /P6+ and NW-BFS-Edge algorithms, of thirty digraphs with the same $n = 200$ cells and m arcs. f shows the average number of edge-disjoint paths of thirty digraphs with the same n cells and m arcs.

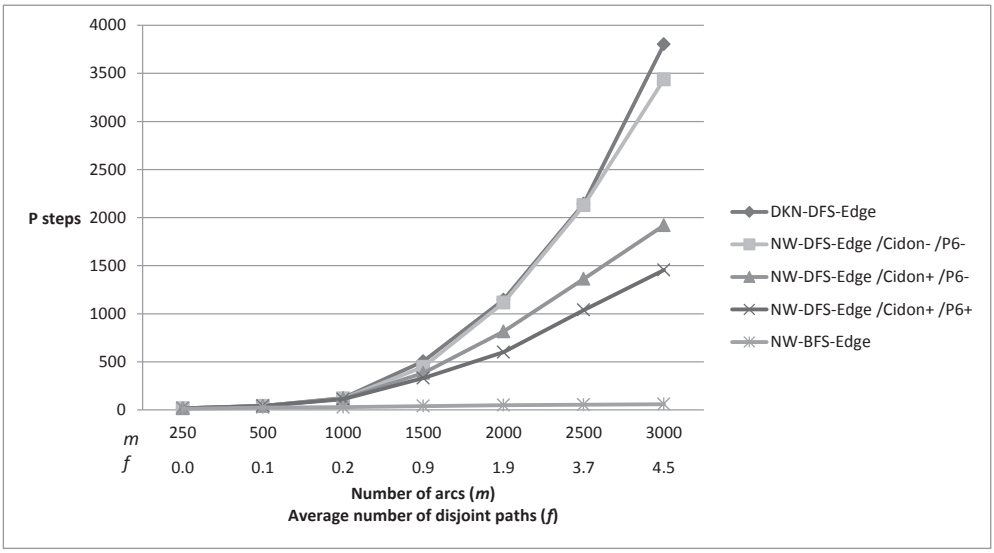


Figure 11: Average performance (P steps) of DKN-DFS-Edge, NW-DFS-Edge /Cidon- /P6-, NW-DFS-Edge /Cidon+ /P6-, NW-DFS-Edge /Cidon+ /P6+ and NW-BFS-Edge algorithms, of thirty digraphs with the same $n = 300$ cells and m arcs. f shows the average number of edge-disjoint paths of thirty digraphs with the same n cells and m arcs.

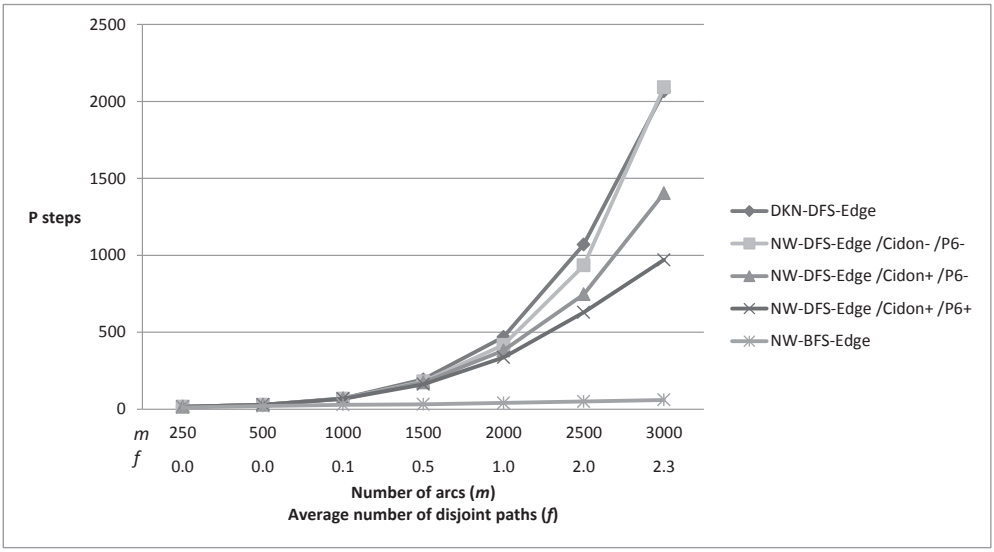


Figure 12: Average performance (P steps) of DKN-DFS-Edge, NW-DFS-Edge /Cidon- /P6-, NW-DFS-Edge /Cidon+ /P6-, NW-DFS-Edge /Cidon+ /P6+ and NW-BFS-Edge algorithms, of thirty digraphs with the same $n = 400$ cells and m arcs. f shows the average number of edge-disjoint paths of thirty digraphs with the same n cells and m arcs.

Table 9: Speed up percentages of NW-DFS-Edge and NW-BFS-Edge over DKN-DFS-Edge, for performance results in Figure 9 ($n = 100$).

	DKN-DFS-Edge	NW-DFS-Edge		NW-BFS-Edge	
m	P steps	P steps	speed up %	P steps	speed up %
250	63	58	7	24	62
500	321	208	35	39	88
1000	1882	748	60	61	97
1500	6396	1683	74	81	99
2000	11364	2386	79	87	99
2500	18817	3201	83	102	99
3000	30189	4262	86	115	100

Table 10: Speed up percentages of NW-DFS-Edge and NW-BFS-Edge over DKN-DFS-Edge, for performance results in Figure 10 ($n = 200$).

	DKN-DFS-Edge	NW-DFS-Edge		NW-BFS-Edge	
m	P steps	P steps	speed up %	P steps	speed up %
250	30	29	3	18	40
500	72	69	4	26	64
1000	380	250	34	36	91
1500	1381	670	51	51	96
2000	3505	1229	65	60	98
2500	7883	2117	73	73	99
3000	11671	2874	75	85	99

Table 11: Speed up percentages of NW-DFS-Edge and NW-BFS-Edge over DKN-DFS-Edge, for performance results in Figure 11 ($n = 300$).

	DKN-DFS-Edge	NW-DFS-Edge		NW-BFS-Edge	
m	P steps	P steps	speed up %	P steps	speed up %
250	16	16	2	12	26
500	43	42	2	25	42
1000	125	111	11	29	77
1500	505	331	35	40	92
2000	1145	600	48	50	96
2500	2144	1099	49	57	97
3000	3802	1454	62	60	98

Table 12: Speed up percentages of NW-DFS-Edge and NW-BFS-Edge over DKN-DFS-Edge, for performance results in Figure 12 ($n = 400$).

	DKN-DFS-Edge	NW-DFS-Edge		NW-BFS-Edge	
m	P steps	P steps	speed up %	P steps	speed up %
250	16	15	5	11	31
500	29	28	3	20	31
1000	70	67	4	28	60
1500	193	161	17	32	83
2000	468	335	28	40	91
2500	1070	629	41	50	95
3000	2065	970	53	60	97

8 Conclusions

We proposed asymptotically faster DFS-based P system algorithms and improved BFS-based P system algorithms for the edge- and node-disjoint paths problems. Empirical results show that, in terms of P steps, our DFS-based algorithms outperform the previous DFS-based algorithms [4]. The performance speed up is based an improved DFS, using (1) Cidon's distributed DFS and (2) a new result, Theorem 6, proposed and proved in this paper, which shows that cells visited in failed rounds can be further ignored, i.e. remain in a permanently visited state.

All our P system algorithms are fixed-size, i.e. each algorithm has a fixed number of rules, which does not depend on the number of cells in the underlying P system. This validates our enhanced definition for P modules, which uses complex symbols and generic rules.

According to their size, our P algorithms seems to compare favourably against any other runnable description. The appendix pseudocodes seem shorter, but these are only high level descriptions, missing quite a few other ingredients required to make them runnable.

Several interesting questions and directions remain open. Can we solve this problem using a restricted P system without states, without sacrificing the current descriptive and performance complexity? What is the expected complexity of our algorithms? How does the speed up depend on the digraph density? Are there other possible optimisations? Finally, another direction is to investigate disjoint paths solutions on P systems with asynchronous semantics, where additional speedup is expected, perhaps a mixed BFS-DFS solution, which combines the advantages of distributed BFS and distributed DFS.

Acknowledgments

The authors wish to thank Tudor Balanescu, Michael J. Dinneen, Hossam ElGindy, Yun-Bum Kim and John Morris, for valuable comments and feedback.

References

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

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

[3] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[4] M. J. Dinneen, Y.-B. Kim, and R. Nicolescu. Edge- and vertex-disjoint paths in P modules. In G. Ciobanu and M. Koutny, editors, *Workshop on Membrane Computing and Biologically Inspired Process Calculi*, pages 117–136, 2010.

[5] M. J. Dinneen, Y.-B. Kim, and R. Nicolescu. A faster P solution for the Byzantine agreement problem. In M. Gheorghe, T. Hinze, and G. Păun, editors, *Conference on Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 175–197. Springer-Verlag, Berlin Heidelberg, 2010.

[6] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[7] A. Eichmann, T. Makinen, and K. Alitalo. Neural guidance molecules regulate vascular remodeling and vessel navigation. *Genes Dev.*, 19:1013–1021, 2005.

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

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

[10] M. Ionescu and D. Sburlan. On P systems with promoters/inhibitors. *Journal of Universal Computer Science*, 10(5):581–599, 2004.

[11] D. C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, New York, NY, USA, 1991.

[12] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[13] R. Nicolescu. Parallel and distributed algorithms in P systems. In M. Gheorghe, G. Păun, G. Rozenberg, A. Salomaa, and S. Verlan, editors, *Membrane Computing, CMC 2011, Revised Selected Papers*, volume 7184 of *Lecture Notes in Computer Science*, pages 35–50. Springer Berlin / Heidelberg, 2012.

[14] R. Nicolescu and H. Wu. BFS solution for disjoint paths in P systems. In C. Calude, J. Kari, I. Petre, and G. Rozenberg, editors, *Unconventional Computation*, volume 6714 of *Lecture Notes in Computer Science*, pages 164–176. Springer Berlin / Heidelberg, 2011.

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

[16] G. Păun, G. Rozenberg, and A. Salomaa. *The Oxford Handbook of Membrane Computing*. Oxford University Press, Inc., New York, NY, USA, 2010.

[17] D. Seo and M. Thottethodi. Disjoint-path routing: Efficient communication for streaming applications. In *IPDPS*, pages 1–12. IEEE, 2009.

[18] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000.

Appendix

This appendix contains high-level pseudocode for sequential versions of the algorithms discussed in this paper. These algorithms share the following definitions:

- $\Pi = (O, V, E)$ is a simple P module and $G = (V, E)$ is its underlying digraph
- $\sigma_s \in V$ is the source cell, and $\sigma_t \in V$ is the target cell
- r is the current round number
- P_{r-1} is the set of 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 $\#r$
- d is the outdegree of the source cell, σ_s
- $\sigma_{s1}, \sigma_{s2}, \ldots \sigma_{sd}$ are the children of σ_s

Pseudocode 1: Classical Edge-disjoint Paths Algorithm

Input : a simple P module $\Pi = (O, V, E)$ and its underlying digraph $G = (V, E)$,
a source cell, $\sigma_s \in V$, and a target cell, $\sigma_t \in V$
$r = 0, P_0 = \emptyset$, $G_0 = G$
repeat
  $r = r + 1$
  $\alpha =$ DFS$(\sigma_s, \sigma_t, G_{r-1})$ // see Pseudocode 2
  if $\alpha =$ **null then break**
  $\overline{P_r} = (\overline{P_{r-1}} \setminus \overline{\alpha}^{-1}) \cup (\overline{\alpha} \setminus \overline{P_{r-1}}^{-1})$
  $G_r = (V, E_r)$, where $E_r = (E \setminus \overline{P_r}) \cup \overline{P_r}^{-1}$
  reset all *visited* cells and arcs to *unvisited*
until $\alpha =$ **null**
Output : P_r, which is a maximum cardinality set of edge-disjoint paths

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

DFS$(\sigma_i, \sigma_t, G_{r-1})$
Input : the current cell, $\sigma_i \in V$, the target cell, $\sigma_t \in V$
and the residual digraph, G_{r-1}
if $\sigma_i = \sigma_t$ **then return** σ_t
if σ_i is *visited* **then return null**
set σ_i as *visited*
foreach *unvisited* $(\sigma_i, \sigma_k) \in E_{r-1}$
  set (σ_i, σ_k) as *visited*
  $\beta =$ DFS$(\sigma_k, \sigma_t, G_{r-1})$
  if $\beta \neq$ **null return** $\sigma_i.\beta$
endfor
return null
Output : a σ_i-to-σ_t path, if any; otherwise, **null**

Pseudocode 3: Classical Edge-disjoint Paths Algorithm—1st DFS call unrolled

Input : a simple P module $\Pi = (O, V, G)$, where $G = (V, E)$,
a source cell, $\sigma_s \in V$, and a target cell, $\sigma_t \in V$
$r = 0, P_0 = \emptyset$, $G_0 = G$
repeat
  $\alpha =$ **null**
  $\beta =$ **null**
  while there is an *unvisited* arc $(\sigma_s, \sigma_q) \in E_{r-1}$ **and** $\beta =$ **null**
    $r = r + 1$
    set σ_s and (σ_s, σ_q) as *visited*
    $\beta = \mathrm{DFS}(\sigma_q, \sigma_t, G_{r-1})$ // see Pseudocode 2
    if $\beta =$ **null then** // failed round
      $G_r = G_{r-1}$
    endif
  endwhile
  if $\beta \neq$ **null then** // successful round
    $\alpha = \sigma_q.\beta$
    $\overline{P_r} = (\overline{P_{r-1}} \setminus \overline{\alpha}^{-1}) \cup (\overline{\alpha} \setminus \overline{P_{r-1}}^{-1})$
    $G_r = (V, E_r)$, where $E_r = (E \setminus \overline{P_r}) \cup \overline{P_r}^{-1}$
    reset all *visited* cells and arcs to *unvisited*
  endif
until $\alpha =$ **null**
Output : P_r, which is a maximum cardinality set of edge-disjoint paths

Pseudocode 4: DKN-DFS-Edge

Input : a simple P module $\Pi = (O, V, G)$, where $G = (V, E)$,
a source cell, $\sigma_s \in V$, and a target cell, $\sigma_t \in V$
$P_0 = \emptyset$, $G_0 = G$
for $r = 1$ **to** d
  set σ_s and (σ_s, σ_{sr}) as *visited*
  $\beta = \mathrm{DFS}(\sigma_{sr}, \sigma_t, G_{r-1})$ // see Pseudocode 2
  if $\beta =$ **null then** // failed round
    $G_r = G_{r-1}$
  else // successful round
    $\alpha = \sigma_{sr}.\beta$
    $\overline{P_r} = (\overline{P_{r-1}} \setminus \overline{\alpha}^{-1}) \cup (\overline{\alpha} \setminus \overline{P_{r-1}}^{-1})$
    $G_r = (V, E_r)$, where $E_r = (E \setminus \overline{P_r}) \cup \overline{P_r}^{-1}$
    reset all *visited* cells and arcs to *unvisited*
  endif
endfor
Output : P_r, which is a maximum cardinality set of edge-disjoint paths

Pseudocode 5: NW-DFS-Edge

Input : a simple P module $\Pi = (O, V, G)$, where $G = (V, E)$,
a source cell, $\sigma_s \in V$, and a target cell, $\sigma_t \in V$
$P_0 = \emptyset$, $G_0 = G$
set σ_s as *permanently visited*
foreach arc $(\sigma_j, \sigma_s) \in E$
  set (σ_j, σ_s) as *permanently visited*
endfor
for $r = 1$ **to** d
  set (σ_s, σ_{sr}) as *permanently visited*
  $\beta =$ Cidon_DFS$(\sigma_{sr}, \sigma_t, G_{r-1})$ // see Pseudocode 6
  if $\beta =$ **null then** // failed round
    $G_r = G_{r-1}$
    set all *temporarily visited* cells and arcs to *permanently visited*
  else // successful round
    $\alpha = \sigma_{sr}.\beta$
    $\overline{P_r} = (\overline{P_{r-1}} \setminus \overline{\alpha}^{-1}) \cup (\overline{\alpha} \setminus \overline{P_{r-1}}^{-1})$
    $G_r = (V, E_r)$, where $E_r = (E \setminus \overline{P_r}) \cup \overline{P_r}^{-1}$
    reset all *temporarily visited* cells and arcs to *unvisited*
  endif
endfor
Output : P_r, which is a maximum cardinality set of edge-disjoint paths

Pseudocode 6: Cidon's DFS, adapted for residual digraph $\mathbf{G_{r-1}} = (\mathbf{V}, \mathbf{E_{r-1}})$

Cidon_DFS$(\sigma_i, \sigma_t, G_{r-1})$
Input : the current cell, $\sigma_i \in V$, the target cell, $\sigma_t \in V$
and the residual digraph, G_{r-1}
if $\sigma_i = \sigma_t$ **then return** σ_t
set σ_i as *temporarily visited*
foreach *unvisited* arc $(\sigma_j, \sigma_i) \in E_{r-1}$
  set (σ_j, σ_i) as *temporarily visited*
endfor
foreach *unvisited* arc $(\sigma_i, \sigma_k) \in E_{r-1}$
  set (σ_i, σ_k) as *temporarily visited*
  $\beta =$ Cidon_DFS$(\sigma_k, \sigma_t, G_{r-1})$
  if $\beta \neq$ **null then return** $\sigma_i.\beta$
endfor
return null
Output : a σ_i-to-σ_t path, if any; otherwise, **null**